

INSIGHT: Atari

Bill Wilkinson
Cupertino, CA

Editor's Note: We're quite pleased to announce a new column this month for Atari owners. INSIGHT: Atari, written by Bill Wilkinson and other staff members of Optimized Systems Software, will bring you monthly programming insight and support.

We feel you'll be quite pleased. — RCL

Hi. I'm Bill Wilkinson, and this is the premiere of what will be a regular feature in **COMPUTE!** magazine: a column dedicated to the *software* side of the Atari microcomputers. We may occasionally include little tricks to make better use of the hardware, but the intent is that this column will uncover the facts and foibles of Atari software.

This column will normally be written by some of the authors of Atari BASIC, Atari's Assembler-Editor, Atari's Disk File Manager, and BASIC A+ and OS/A+. We are not all experts in Atari hardware, but we know a lot about the software.

Addressable DATA or Who Needs String Arrays?

Perhaps the most frequent complaint made about Atari BASIC pertains to its lack of string arrays. In 10K bytes of ROM one can pack only so much program; long variable names and instant syntax checking take room; HP and DG have very successful BASICs that don't use string arrays; Atari-style strings are fast and flexible. All this doesn't mean much to you if you can't figure a way to convert that neat Applesoft program to Atari. There are many legitimate uses of string arrays, but the most common use is a kind of in-memory random access data file. Example: in an adventure game the various room descriptions are kept in elements of a string array. This is not the fullest exploitation of string arrays, since the data is static and the arrays merely provide a convenient method of addressing it.

Atari BASIC users, take heart! You have available to you an even more powerful and flexible method of randomly addressing static data. Did you ever notice that Atari BASIC supports the syntax "RESTORE line-number"? Did you ever notice that "line-number" can be either a constant number or (surprise) *any* arbitrary numeric expression? These two capabilities combine to allow some extremely powerful programming constructs in Atari BASIC. The following short program will serve to illustrate.

Let us go through this program carefully and search out the tricks. Lines 1000-1030 are fairly straightforward; the variable names were purposefully chosen to demonstrate that Atari BASIC considers *all* characters in a name to be significant. Lines 1100-1120 initialize the variables which will

```

1000 REM a demonstration of addressable DATA
1010 REM allocate some variables
1020 DIM ROOM$(100),GOS$(1),DIRECTION(4),
    DIRECTION$(4)
1030 LET DIRECTION$="NESW"
1100 REM the following variables are used as line
    numbers, etc.
1110 LOOKROOM=3000:LOOP=2000:
    DESCRIPTIONS=9000
1120 DESCRIPTIONSIZE=10
1900 REM variables are set up — initialize player
    status
1910 ROOM=2:GOSUB LOOKROOM
2000 REM the main program loop
2010 PRINT "WHICH WAY?";:INPUT GOS
2020 DIRECTION=0
2030 FOR I=1 TO 4:IF GOS=DIRECTION$(I,I)
    THEN DIRECTION=I
2040 NEXT I
2050 IF NOT DIRECTION THEN GOTO LOOP
2060 GO=DIRECTION(DIRECTION)
2070 IF NOT GO THEN PRINT "CAN'T GO THAT
    WAY":GOTO LOOP
2080 IF GO>1000 THEN GOSUB GO:GOTO LOOP
2090 ROOM=GO:GOSUB LOOKROOM
2100 GOTO LOOP
3000 REM subroutine to get and print details of a
    new room
3010 RESTORE DESCRIPTIONS+ROOM*
    DESCRIPTIONSIZE
3020 FOR I=1 TO 4:READ TEMP:DIRECTION(I)
    =TEMP:NEXT I
3030 READ ROOM$:PRINT "YOU ARE IN";
    ROOM$
3040 RETURN
8000 REM special routines for special actions
8010 PRINT "YOU MADE IT OUT! CONGRATU-
    LATIONS!":END
9000 REM the room descriptions and connections
9010 DATA 3,5,0,0,A LARGE CAVERN
9020 DATA 0,4,5,3,A SMALL CAVERN
9030 DATA 0,2,1,0,A CURVING PASSAGEWAY
9040 DATA 0,8010,5,2,AN ANTECHAMBER
9050 DATA 2,4,0,1,A MAZE OF TUNNELS

```

be used for "address arithmetic" later in the program; "LOOP," for example, simply gives a name to the line number where all the action starts.

Line 1910 begins the start of the tricks: it GOSUBs to LOOKROOM. Notice how much more readable this is than simply coding GOSUB 3000, which tells you nothing of the purpose of the statement. Looking at routine LOOKROOM (lines 3000-3040), we note the usage of "RESTORE expression." As an example, assume that LOOKROOM is called with ROOM=2. Then line 3010 becomes equivalent to "RESTORE 9020." The subsequent READs then fill the array DIRECTION() with the numeric data of line 9020 and the string ROOM\$ with the string, "A SMALL CAVERN." Finally, the user is prompted with a message ("YOU ARE IN A SMALL CAVERN,") and the subroutine exists.

Continuing our main program at lines 2000-2040, we simply ask the user for a direction (from the choices 'N', 'E', 'S', and 'W'). An invalid answer

causes DIRECTION to be zero and the question to be asked again. Let us assume we are still in room two and also assume that the WHICH WAY? query was answered by "E." GO then becomes 4 (from DIRECTION(2)); and, since it is nonzero (line 2070) and less than 1000 (line 2080), the current ROOM becomes number 4 and we GOSUB LOOK-ROOM again.

The only things left to note about this program are what happens if GO is zero (e.g., if we had tried to go "N" from room 2) or greater than 1000 (if we try to go "E" from room 4)? The case of GO=0 is easy: the program treats that as an illegal move, prints "CAN'T GO THAT WAY," and makes the player try again. For GO greater than 1000, another action unique to Atari BASIC happens: GOSUB to the apparent room number contained in GO. In the particular example shown, the only GOSUB is to line 8010 which ends the "adventure," but this mechanism can be used to allow sophisticated checks on movement (e.g., you can only go from room 31 to room 33 if you have the Golden Fleece). The concept of addressable GOSUBs was heavily exploited, and we will try to cover those techniques in a future column.

Each of these columns will cover one or two programming topics and answer a few questions (presuming that you, the reader, will supply us with some questions). In this initial column, we would like to try to comment on some of the points raised in the "ASK THE READERS" column from **COMPUTE! #14**.

16K Memory

I. Regarding D. Gallagher's query about PRINT FRE(0) in his 48K machine.

When you plug the first (left, in an Atari 800) cartridge into an Atari, you "lose" the top 8K of the possible 48K of RAM. Thus your 48K does you no more good than 40K would. It can get worse: if Atari ever comes out with a dual cartridge product, you will lose the top 16K of your 48K. The reason: Atari's memory map simply doesn't leave any other place to put the cartridges, so Atari cleverly arranged the circuitry so that plugging in the cartridge disables any RAM at the same addresses. Does this mean that it is a waste to put 48K bytes of RAM into your Atari? Not at all! There are several products already available that use no cartridges at all (Visicalc, BASIC A+, Forth, etc.). In fact, look for Atari systems with 160K bytes of RAM, or more, in the near future. And by the way, it is not surprising to hear of the "foreign" memory board in the Atari: systems suppliers have been doing that in the minicomputer (DEC, HP, etc.) and S-100 (8080 and Z-80) markets for years! After all, if the dealer can give you more for less, why complain? Oh yes, for the curious, herewith the Atari memory map:

OS ROM (10K bytes)		FFFF
I/O hardware		D800
Reserved for future use		D000
48K of RAM		C000
	Left cartridge	A000
	Right cartridge	8000
		0000

II. Comments about the letter discussing RFI from an APPLE II.

Atari owners, stand up and be proud! Did you know that your machine is the only full-fledged computer that was able to pass the FCC's former (and very strict) RFI regulations? But thanks to TI, and some extensive lobbying with the FCC, the RFI rules are much relaxed and even the Apple II (with the help of some new shielding) can now pass the tests. But even so, the Atari has to be one of the quietest (in terms of RFI) machines ever produced. So while you owners are enjoying noise-free television, remember that the abysmally slow disk I/O speeds you also "enjoy" are part of Atari's solution to the RFI problem. That serial bus didn't just happen by accident: it was the result of some superb — but, alas, no longer necessary — engineering.

III. An answer to Tracy Principio about GR.X from assembly language.

Anyone contemplating writing in assembly language for the Atari is virtually required to purchase the *Hardware Manuals* (as did Tracy); but, even if you don't have a disk, the Atari DOS manuals and OS listings are *also* de rigueur. Any kind of I/O must go through CIO, the heart of the Atari OS, and graphics on the Atari are most easily done via I/O. Did you know that PLOT, DRAWTO, POSITION, FILL, and more are *not* in Atari Basic? They are actually routines in the I/O section of the OS ROM, and BASIC simply provides an interface to them. So, if you understand the I/O subsystem, you can do graphics in assembly language almost as easily as you can do them in BASIC. The whole subject of I/O and graphics from assembly language would make a beautiful *series* of columns (tell us if you'd like to see some), so we must "answer" Tracy's question by noting that GR.X is equivalent to:

OPEN #6,12,X,"S:" if X is greater than 16
(full screen graphics)
or OPEN #6,12+16,X,"S:" if X is less than 16
(mixed characters and graphics).

Note that the "12" is simply 8+4, read *and* write access, just as with a disk.

That's all for this month. We hope that by increasing your awareness of its capabilities we can convert you, too, into more informed and capable Atari users. ©

Insight: Atari

Bill Wilkinson
Optimized Systems Software
Cupertino, CA

Last month we explored the possibilities inherent in the fact that Atari BASIC supports addressable DATA statements. This month we will tackle a related subject. It will be fairly obvious that the techniques presented in these two articles can help one write a fast yet complex adventure game on the Atari. However, it would be most interesting to me to see what other uses you readers make of these ideas, so start those cards and letters coming!

Nonexistent Subroutines ... Or Having Fun While GOSUBing Nowhere

Purists should probably not read this section: we will plumb depths that advocates of structured programming would never sink to. Just as Atari BASIC allows the code `RESTORE <expression>`, we can also use `GOTO <expression>` and `GOSUB <expression>`. (Don't worry about the notation: `<anything>` just means that "anything" is an English language word instead of a reserved BASIC word.) Allowing `GOTO` and `GOSUB` to refer to arbitrary expressions instead of absolute line numbers is unusual in BASICs (and well nigh impossible in most other languages), so perhaps the power of this capability has never been fully realized. We will try to make a few inroads.

Before we get into the more exotic part of our discussion, let us note the most obvious advantage of "line number expressions": self-documenting code. How much more meaningful it is to be able to code `GOSUB CALCULATEPAY` instead of `GOSUB 13250` !! Admittedly, with Atari BASIC one must first have written `LET CALCULATEPAY = 13250`; but that is a small price to pay for the added readability. Fair warning: there is one drawback to this trick. Atari BASIC allows only 128 different variable names. Normally that is a very big number, but naming every subroutine or section of code can eat up variables in a hurry. Be judicious in your choice of which routines are worth naming.

Now it is time for our main topic. And, thus, time for an example program. Study the example carefully before continuing with this text.

Lines 1000-1030 are simply set-up and initialization, one time operations. Lines 2000-2260 constitute the main loop of the program (in fact, in this simplistic example, this is an endless loop). First, the user is asked for a verb. If the `INPUT` verb matches one in the DATA list, it is assigned an appropriate verb number. The process is repeated for a noun. In an actual adventure game, the user would presumably be asked for "VERB NOUN" via a single `INPUT`; the program would then have to parse the request into the separate words.

At line 2200-2260 we exhibit the trick that this article is all about. To understand what happens, let us follow through what would happen if the user had requested "KISS BABY." "KISS" is verb number 2 and "BABY" is noun number 3, so at line 2220 we attempt to `GOSUB` to line $10000 + 2 * 1000 + 3 * 10$; that is we attempt to `GOSUB 12030`. Lo and behold, line 12030 causes the message "YOU HAVE JUST BEEN ELECTED MAYOR" to print out. When the routine `RETURNS`, we `GOTO LOOP` and do this all over again.

But wait: suppose the user had typed "LOOK BABY." That phrase evaluates to verb 1 and noun 3, so we try to `GOSUB 11030` ($10000 + 1 * 1000 + 3 * 10$). But there is no line 11030! All is not lost: note that on line 2210 we first `TRAPPED` to line 2250. The attempt to `GOSUB 11030` will trigger the `TRAP` and we indeed will continue execution at line 2250. Here, we attempt to `GOSUB` to line $10000 + 1 * 1000$, effectively ignoring the noun. We succeed in executing line 11000, the "default" routine for the verb "LOOK," and find that the computer sees "NOTHING SPECIAL" about this baby.

The power implicit here is perhaps not obvious. But consider how easy it is to add new verbs and nouns to this program. Consider how easy it is to provide for as many or as few special verb-noun combinations as you wish. And, finally, look how little code is used!

Note that this program expects there will be a routine for each valid verb (it's only sensible: why have a verb in the DATA if it doesn't do anything)? Another `TRAP` statement, at line 2250, could allow for omitted verbs. By the way, with the program written as it is, there is no way to get to line 2250 with the error `TRAP` system still active. Atari BASIC always resets any `TRAP` when it is triggered (this is so that you can't accidentally fall into endless `TRAP` loops).

The techniques discussed in this and prior articles have actually been used to write a "PICO-ADVENTURE." The most amazing aspect of the program is the speed with which it responds: it seems as fast or faster than even machine language adventures. Try it. Let us know about your efforts.

Unreadable Programs

There follow two program lines that can be added to any existing Atari BASIC program and which, when executed, will make a program virtually unLISTable. The first line simply changes the names of all your variables to RETURN characters, and can be used with or without the second line. The second line actually produces a BASIC SAVED program that can only be RUN — not LISTed, LOAded, etc.

Atari BASIC version:

```
32766 FOR I=PEEK(130)+256*PEEK(131) TO
      PEEK(132)+256*PEEK(133): POKE I,155:
      NEXT I
32767 POKE PEEK(138)+256*PEEK(139)+2,0: SAVE
      "<filename>": NEW
```

BASIC A+ version:

```
32766 for i=dpeek(130) to dpeek(132): poke i,155:
      next i
32767 poke dpeek(138)+2,0: save "<filename>"
      : new
```

To use these gems, simply enter them and then type GOTO 32766. The line numbers are not important, but the second line must be the last line of the program. To use the resulting program, simply type RUN "<filename>". The program should *not* end with STOP or END; instead, it should exit via NEW (yes, "NEW" can be used from within a program). The <filename> may be "C:", but CLOAD will not work (you must use RUN"C:").

VARIABLE, VARIABLE, VARIABLE

Perhaps one of the more common mistakes when using the long variable names allowed by Atari BASIC is to make a typo when entering the name (I tend to leave off the plural, "s"). How to know you have committed this sin? Try the following program segment:

Using Atari BASIC:

```
32700 I=0: FOR J=PEEK(130)+256*PEEK(131) TO
      PEEK(132)+256*PEEK(133)-1
32710 IF PEEK(J) < 128 THEN PRINT CHR$(PEEK
      (J)): GOTO 32730
32720 PRINT CHR$(PEEK(J)-128): I=I+1
32730 NEXT J: PRINT: PRINT I; " VARIABLES
      IN USE": STOP
```

Using BASIC A+:

```
32700 i=0: for j=dpeek(130) to dpeek(132 + 1: print
      chr$(peek(j) & 127);
32710 if peek(j) > 128: print: i=i+1: endif: next j
32720 print: print i; " variables in use": stop
```

it into any program that needs it. To use, simply type GOTO 32700; all your variables will be listed and a count displayed. Obviously, this output could be sent to the printer by first OPEN #7,8,0, "P:" and then replacing all the PRINTs with PRINT #7. If you do this, it is advisable to CLOSE #7 before the STOP.

```
1000 REM ***** SET UP *****
1010 DIM VERB$(4),NOUN$(4),TEST$(4)
1020 VERB$DATA=9000:NOUN$DATA=9100
1030 LOOP=2000
2000 REM ***** MAIN LOOP *****
2010 PRINT "GIVE ME A VERB ";:INPUT VERB$
2020 RESTORE VERB$DATA:VERB=0
2030 FOR CNT=1 TO 3:REM CHANGE TO MATCH DATA
2040 READ TEST$
2050 IF TEST$=VERB$ THEN VERB=CNT:CNT=99
2060 NEXT CNT
2070 IF NOT VERB THEN PRINT "INVALID VERB":
      GOTO LOOP
2100 REM VERB DONE, DO NOUN
2110 PRINT "GIVE ME A NOUN ";:INPUT NOUN$
2120 RESTORE NOUN$DATA:NOUN=0
2130 FOR CNT=1 TO 3:REM CHANGE TO MATCH DATA
2140 READ TEST$
2150 IF TEST$=NOUN$ THEN NOUN=CNT:CNT=99
2160 NEXT CNT
2170 IF NOT NOUN THEN PRINT "INVALID NOUN":
      GOTO LOOP
2200 REM ***** THE TRICKY STUFF *****
2210 TRAP 2250
2220 GOSUB 10000+VERB*1000+NOUN*10
2230 GOTO LOOP
2240 REM WE GET TO 2250 ONLY ON TRAP
2250 GOSUB 10000+VERB*1000
2260 GOTO LOOP
9000 REM A LIST OF ALL VERBS
9010 DATA LOOK,KISS,DROP
9100 REM A LIST OF ALL NOUNS
9110 DATA ROOM,BEAR,BABY
10000 REM *****
10010 REM * THE VERB-NOUN ACTION *
10020 REM *****
11000 REM >>> LOOK <<<
11001 PRINT "I SEE NOTHING SPECIAL":RETURN
11010 PRINT "I SEE A WINDOW AND A DOOR":RETURN
12000 REM >>> KISS <<<
12001 PRINT "THAT'S SILLY...BUT SMACK":RETURN
12020 PRINT "BEAR BITES OFF YOUR LIPS":RETURN
12030 PRINT "YOU HAVE JUST BEEN ELECTED
      MAYOR":RETURN
13000 REM >>> DROP <<<
13001 PRINT "HOW? I COULDN'T HAVE LIFTED
      THAT.":RETURN
13030 PRINT "IT'S A BOUNCING BABY BOY!!"
      :RETURN
```

©

TOLL FREE
Subscription
Order Line
800-345-8412
 In PA 800-662-2444

It would be advisable to LIST this program segment to DISK or CASSETTE and then ENTER

Insight: Atari

Bill Wilkinson
Cupertino, CA

In my September column, I mentioned that the subject of graphics and I/O would make a nice series of columns. I wondered if there would be enough interest in the topic to justify the writing effort. Since that time, I have (in the course of talking to our customers) discovered that not only is there interest in the topic, but there is also a woeful lack of information and an abundance of misinformation regarding Atari's OS. So, with this column, we start a three or four part series on assembly language I/O.

Also, this month's column includes a list of major, known bugs in Atari BASIC and how to get around them.

Atari I/O, Part One: Interfacing To OS

Before I get started with the hairy details, I would like to state that Atari has the *best operating system* in the low-end microcomputer market. There is a simple reason for this: Atari has the *only* operating system on the market! Now, admittedly, I am being a purist when I make this contention, but the truth is that the Atari is the only machine I know of that has a *true* operating system in ROM. And, no, neither my company (Optimized Systems Software) nor I were involved in the creation of that operating system; the credit must go straight to Atari.

The operating system is contained in ROM and is identical on both the Atari 400 and Atari 800. The 10K bytes of ROM you may have noticed contain not only the operating system, but also the upper/lower case character set, the floating point mathematical operations, the power-on and cartridge select logic, and the device drivers. Device drivers? Aren't those part of the operating system? No! An emphatic "no." And that's what enables me to say that Atari has the only true operating system.

Believe it or not, the operating system on the Atari occupies less than 700 bytes. And yet it is as complete in its own way as UNIX is on a large time sharing machine. How many times have you read a magazine and seen lists of addresses of I/O subroutines for XYZ computer? You must use this address to output a character to the screen, another to get a character from the keyboard, yet another to talk to the line printer, and disk I/O? A nightmare! Not so Atari. One and only one address need be remembered: Hex E456, Decimal 58454. (Yes, I know, why not E400 or F000 or some such. Well, I didn't say Atari was perfect, only good.)

With only one address that matters, you can imagine that it should be easy for Atari to come out with new versions of the OS without affecting any

other programs. They have, and they are, and only programs that have "cheated" (gone outside the OS rules) are in trouble. So, don't get yourself in trouble; follow the OS rules.

Finally, to avoid duplication of effort, I would refer you to the massive program listing for "SHOOT" in **COMPUTE!** #16: the first two pages and first column of the next two pages constitute most of the useful equates when using the Atari

**...the operating system
on the Atari occupies
less than 700 bytes.**

from assembly language. We are concerned with the "Operating System Equates" (first column, second page) and "Page three RAM assignments" (third column, first page). I will use the mnemonics given in that listing throughout this series of articles. (Those of you who own our "OS/A+" system will find these equates, with some mnemonics altered slightly for consistency, in the file "SYSEQU.ASM". You may use the EASMD pseudo-op ".INCLUDE #D:SYSEQU.ASM" to include them in any assembly programs. This will save you some typing.)

The Structure Of The IOCB's

When a program calls the OS through location \$E456, OS expects to be given the address of a properly formatted IOCB (Input Output Control Block). For simplicity, Atari has predefined eight IOCB's, each 16 bytes long, and the program specifies which one to use by passing the IOCB number times 16 in the 6502's X-register. Thus, to access IOCB number four, the X-register should contain \$40 on entry to OS. Notice that the IOCB number corresponds directly to the file number in BASIC (as in PRINT #6, etc.). Actually, the IOCB's are located from \$0340 to \$03BF (refer to the "SHOOT" listing).

When OS gets control, it uses the X-register to inspect the appropriate IOCB and determine just what it was that the user wanted done. Table 1 gives the Atari standard names for each field in the IOCB along with a short description of the purpose of the field. Study the Table before proceeding.

The user program should *never* touch fields ICHID, ICDNO, ICSTA and ICPTL/ICPTH. In addition, unless the particular device and I/O request requires it, the program should not change ICAX1 through ICAX6. The most important field is the one-byte command code, ICCOM, which tells the operating system what function is desired.

The OS itself only understands a few fundamental commands, but Atari wisely provided for extended commands necessary to some de-

vices (XIO in BASIC). In any case, each one of these fundamental commands deserves a short description.

OPEN Open a device (synonyms: file, IOCB, channel) for read and/or write access. OS expects ICAX1 to contain a byte that specifies

...the drivers account for over 5K bytes of the ROM code. The screen handler, with all its associated editing and Graphics modes, occupies about 3K bytes of that.

the mode of access: ICAX1 = 4 for read access, 8 for write access, and 12 for both read and write access. (Note: the disk file manager and the screen device handler allow other modes, and they will be discussed in a later section.) The name of the device (and, for the disk, the file) must be given to OS; this is accomplished by placing the ADDRESS of a string containing the name in ICBAL/ICBAH.

CLOSE Terminate access to a device/file. Only the command must be given.

STATUS Request the status of a device/file. The device can interpret this request as it wishes, and pass back a (hopefully) meaningful status. As with OPEN, the ADDRESS of a filename must be placed in ICBAL/ICBAH.

GET TEXT A powerful command, this causes the OS to retrieve ("GET") bytes one at a time from a device/file already OPENed until either the buffer space provided by the user is exhausted or an Atari RETURN character (Hex 9B, Decimal 155) is encountered. The user specifies the buffer to use by placing its ADDRESS in ICBAL/ICBAH and its size (length) in ICBLL/ICBLH.

PUT TEXT The analog of GET TEXT, OS outputs characters one at a time until a RETURN is encountered or the buffer is empty. Requires ICBAL/ICBAH and ICBLL/ICBLH to be specified.

GET DATA Extremely flexible command, this causes OS to retrieve, from the device/file previously OPENed, the number of bytes specified by ICBLL/ICBLH into the buffer specified by ICBAL/ICBAH. *No checks whatsoever are performed on the contents of the transferred data.*

PUT DATA Similar to GET DATA, except that OS will output ICBLL/ICBLH bytes from the buffer specified by ICBAL/ICBAH. Again,

no data checks are performed.

Table 2 provides the OS commands and their usage of the various fields of the IOCB's. For convenience, the disk file manager extended commands are also shown, but I must withhold discussion of them until next month.

Device names on the Atari computers are very simplistic; they consist of a single letter (optionally followed by a single numeral). Traditionally (and, in the case of disk files, of necessity) the device name is followed by a colon. You have probably seen these device names in your various Atari manuals, but a quick summary might be convenient:

- E:** The keyboard/screen editor device. The normal console output.
- K:** The keyboard alone. Use this device to bypass editing of user input.
- S:** The screen alone. Can be either characters (à la E:) or graphics.
- P:** The printer. The standard device driver allows only one printer.
- C:** The cassette recorder.
- D:** The disk file manager, which also usually requires a file name.

Other device names are possible (e.g., for RS-232 interfaces) and, in fact, the ease with which other devices may be added is another reason for my claim that Atari has a *true* operating system. The structure of device drivers is material for a later article, but I should like to point out that the OS ROM includes drivers for all the above except the disk. In fact, the drivers account for over 5K bytes of the ROM code. The screen handler, with all its associated editing and Graphics modes, occupies about 3K bytes of that.

Actually, the next column will begin to delve deeper into the ways of using OS, but for those of you anxious and brave enough to get started now we present a very simple example program:

PUTMSG		; A ROUTINE TO PRINT A MESSAGE
LDX	#\$00	; WE USE IOCB NUMBER 0, THE CONSOLE (E:)
LDA	#PUTREC	
STA	ICCOM,X	; THE COMMAND IS 'PUT TEXT RECORD'
LDA	#MSG&255	
STA	ICBAL,X	; LOWER BYTE OF ADDRESS OF 'MSG'
LDA	#MSG/256	
STA	ICBAH,X	; UPPER BYTE OF ADDRESS
LDA	#255	
STA	ICBLL,X	; LOWER BYTE OF LENGTH OF MSG
STA	ICBLH,X	; UPPER BYTE, LENGTH IS ALL OF MEMORY
JSR	CIOV	; BUT 'PUTREC' WILL STOP WITH THE 'RETURN' CHAR
TYA		; CALL THE OS TO DO THE WORK
BMI	ERROR	; MOVES RETURNED ERROR CODE TO A-REGISTER
		; ANY NEGATIVE VALUE IS SOMESORT OF ERROR

Table 1.

FIELD NAME	OFFSET WITHIN IOCB (BYTES)	SIZE OF FIELD (BYTES)	PURPOSE OF FIELD
ICHID	0	1	SET BY OS. Index into device name table for currently OPEN file, set to \$FF if no file open on this IOCB.
ICDNO	1	1	SET BY OS. Device number (e.g., 1 for "D1:xxx" or 2 for "D2:yyy")
ICCOM	2	1	The COMMAND request from user program. Defines how rest of IOCB is formatted.
ICSTA	3	1	SET BY OS. Last status returned by device. Not necessarily the status returned via STATUS command request.
ICBAL ICBAH <i>JCBADR</i>	4	2	BUFFER ADDRESS. A two byte address in normal 6502 low/high order. Specifies address of buffer for data transfer or address of file-name for OPEN, STATUS, etc.
ICPTL ICPTH	6	2	SET BY OS. Address minus one of device's put-one-byte routine. Possibly useful when high speed single byte transfers are needed.
ICBL ICBLH <i>JCBLEN</i> <i>u</i>	8	2	BUFFER LENGTH. Specifies maximum number of bytes to transfer for PUT/GET operations. Note: this length is decremented by one for each byte transferred.
ICAX1	10	1	Auxiliary byte number one. Used in OPEN to specify kind of file access needed. Some drivers can make additional use of this byte.
ICAX2	11	1	Auxilliary byte number two. Some serial port functions may use this byte. This and all following AUX bytes are for special use by each device driver.
ICAX3 ICAX4	12	2	For disk files only: where the disk sector number is passed by NOTE and POINT. (These bytes could be used separately by other drivers.)
ICAX5	14	1	For disk files only: the byte-within-sector number passed by NOTE and POINT.
ICAX6	15	1	A spare auxilliary byte.

...
...
; CONTINUE WITH MORE CODE

MSG .BYTE 'THIS IS A MESSAGE', \$9B

Just a very few notes on this routine: (1) If the command had been "GETREC," the OS would have gotten a line from the keyboard and put it into the "buffer" at MSG. (2) If the X-register had been set to \$20 and if the printer had previously been OPENed at IOCB number 2, then *this same code* would have sent the message to the printer. (3) If the buffer length had been given as less than 18, the message would have been truncated to the specified length. That's all on I/O for this month. I hope you will hound your mailbox until your next issue of **COMPUTE!** arrives.

Bugs In BASIC

Several people have requested a list of all known bugs in Atari BASIC. The following list may not be complete, but it certainly enumerates all the bugs that may be considered "killers."

1. In the course of editing a BASIC program, sometimes the system loses all or part of the program and/or simply hangs. Often, turning power off and back on is the only solution. Contrary to popular belief, this condition is related to nothing except the size of the program that is being moved by a delete operation (*not* the size of the deleted line). FIX: NONE. Sorry about that. Just be sure and SAVE your programs often, especially if you are doing heavy editing.
2. String assignments that involve the movement of multiples of 256 bytes do not move the first 256 bytes. FIX: don't move multiples of 256 bytes. An easy way to accomplish this is to always move an ODD number of bytes. Usually, moving one extra byte is fairly easy to handle.
3. The cassette handler doesn't always properly initialize its hardware interface. Symptoms: ERROR 138 and ERROR 143. FIX: use an LPRINT before doing a CSAVE, etc. (This isn't a BASIC bug, but BASIC can be used to fix it.)
4. Taking the unary minus of a zero number (e.g., PRINT -0) can result in garbage. Usually this garbage will not affect subsequent calculations, but it does print strangely. FIX: don't use the unary minus in cases where there may be a doubt (e.g., use PRINT 0-x if 'x' might be zero).
5. Strange things can happen if you type in a program line longer than three screen lines long. Reason: the system editor device (E:) cuts off your input at three lines and gives it to BASIC, which processes it as is, and then E: gives the rest of your input to BASIC as the next line! FIX: don't try to put in program

lines bigger than three screen lines.

6. Using an INPUT statement without a variable (i.e., just '10 INPUT') does *not* cause a syntax error (it should) and may cause program lock-up when RUN. FIX: don't do it. (What did you expect? BASIC is in ROM, so *it* can't be fixed.)

7. Most keywords can be used as variable names. (Try this sometime: LET LET = 5 : LET PRINT = 3 : PRINT PRINT : PRINT LET ... it works!) Some cannot, and BASIC will tell you about them. But 'NOT' cannot be the first three letters of any variable name. Example:

10 LET NOTE = 5 : PRINT NOTE

If you enter that line and then LIST it, you will get

10 LET NOTE = 5 : PRINT NOT E

because in an expression NOT is a unary operator that is never seen as part of a variable name. (In the LET, only a variable name is expected, so NOT is never seen.) This is the only "poison" keyword in Atari BASIC. (Note the use of 'LET' in several instances above. Generally, assignment to a variable name which starts with a keyword requires the use of LET to avoid confusing the syntaxer.)

8. LOCATE and GET do not reinitialize their buffer pointer, so they can do nasty things to

memory if used directly after some statements (e.g., they can change the line number of a DATA statement if used after a READ). FIX: reinitialize the pointer by using a STR\$ function call (e.g., XX = STR\$(0) works fine). Clumsy, but it works. PRINTing a numeric value works also (since PRINT calls STR\$ internally). This fix is probably one you can ignore until it happens to you.

9. An INPUT of more than 128 bytes (from disk, cassette, etc.) will write into the lower half of page six RAM (\$0600-\$0657F). This is *not* a bug, it was designed that way. The lower half of page six was supposed to be available to BASIC, but someone at Atari forgot to tell someone else at Atari (and even two different memory maps in the Atari BASIC Reference Manual don't agree). As a consequence, both Atari and user programmers have come to regard all of page six as their own and have put small assembly language programs there. FIX: don't use the programs from \$0600-\$067F or don't INPUT such long strings.

There are a few other minor bugs (e.g., you can say DIM A(32766,32766) without getting an error message), but, by and large, they won't affect most programs. If anyone thinks they know of any other major bugs, let me know and I will try to provide a fix. Please let us know what topics you want covered.

IOCB field offset and name	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
Type of command	ICHID	ICDNO	ICCOM	ICSTA	ICBAL	ICBAH	PUT-A-BYTE ADDRESS		BUFFER LENGTH		ICAX1	ICAX2	ICAX3	ICAX4	ICAX5	ICAX6	Mnemonic label used by Atari for this field
OPEN	*	*	3	*	FILENAME		*	*			SEE TEXT						OPEN
CLOSE	*		12	*													CLOSE
DYNAMIC STATUS		*	13	*	FILENAME												STATIS
GET TEXT RECORD			5	*	BUFFER				LENGTH								GETREC
PUT TEXT RECORD			9	*	BUFFER				LENGTH								PUTREC
GET BINARY RECORD			7	*	BUFFER				LENGTH								GETCHR
PUT BINARY RECORD			11	*	BUFFER				LENGTH								PUTCHR
EXTENDED COMMANDS: DISK FILE MANAGER ONLY																	
RENAME		*	32	*	FILENAME												RENAME
ERASE		*	33	*	FILENAME												DELETE
PROTECT		*	35	*	FILENAME												LOCKFL
UNPROTECT		*	36	*	FILENAME												UNLOCK
NOTE			38	*									SECTOR NUMBER		BYTE		NOTE
POINT			37	*									SECTOR NUMBER		BYTE		POINT

—LEGEND—

* — Set by OS when this command is used.
 BUFFER — 16-bit address of a data buffer.
 FILENAME — 16-bit address of a filename.
 LENGTH — length (in bytes) of a data buffer.
 SECTOR NUMBER and BYTE — see text.



INSIGHT: Atari

I Wilkinson
Optimized Systems Software
Berkeley, CA

Last month, we tackled some of the fundamentals of I/O under Atari's OS. This month we will look at the extended disk operations available and will try our hand at writing a useful program in assembly language.

There simply isn't space to repeat the charts given in last month's article, so you will have to refer to those pages: we will be referring to them often.

Part I/O, Part 2: Disk File Manager

Notice that the title of this section is *not* "ATARI OS." There is a simple reason, which I expounded on before: Atari does not have a DOS. (But please don't tell them I said so; they think they have to call it "DOS," because that's what everybody else calls it.) Atari *has* an "OS"; actually a much more powerful system than what is normally called "DOS" on microcomputers. And please recall from last month that the Atari OS understands *named* devices, such as "P:" and "E:". The Disk File Manager (DFM) is actually simply a device driver for the disk ("D:") device. It was written completely separately from the Atari OS and interfaces to OS the same way any other driver does. In fact, there is nothing magical about the DFM. In theory, by the end of next month's article you should know enough about the Atari OS and the DFM to implement your own File Manager and to replace the one that Atari supplies you. (*In theory*. In practice, you had better know the principles of disk space allocation, I/O blocking and deblocking, and much more, before tackling such a job.) Even if you aren't quite that ambitious, we hope that this series will give you some "insight" into how such things as BASIC's I/O are implemented.

Extended Disk Operations

We should first note that most of the extended disk

operations are documented in the *Atari Basic Reference Manual* in the section about the XIO command. There are two exceptions, NOTE and POINT, which were given special BASIC commands (and we will see why soon). Naturally, the *Atari Disk Operating System II Reference Manual* is pertinent, but it doesn't really give more information about the internal workings of Atari's OS than does the BASIC manual. Before delving into assembly language, let's examine each of the extended disk operations in a little detail:

ERASE, PROTECT, UNPROTECT — Also known as Delete, Lock, and Unlock, these three commands simply provide OS with a channel number (i.e., the X-register contains IOCB number times 16), a command number (ICCOM), and a filename (via ICBAL/ICBAH). When OS passes control to the DFM, an attempt is made to satisfy the request. Note that the filename may include "wild cards," as in "D:*.*?" (which will affect all files on disk drive one which have an 'S' as the last letter of their filename extension).

RENAME — Very similar to ERASE, et al, in usage. The only difference is in the form of the filename. Proper form is:

"Dn:oldname.ext,newname.ext"

Note that the disk device specifier is not and *cannot* be given twice.

NOTE, POINT — Other than OPEN, these are the only commands we have encountered so far (including last month) which use any of the AUXilliary bytes of the IOCB. For these commands, one specifies the channel number and command number and then receives or passes file pointer information via three of the AUX bytes. ICAX3/ICAX4 are used as a conventional 6502 LSB/MSB 16-bit integer: they specify the current (NOTE) or the-to-be-made-current (POINT) sector within an already OPENed disk file. ICAX5 is similarly the current (NOTE) or to-be-made-current (POINT) byte within that sector. These are complex commands to use, but their operation from BASIC is adequately covered in the *Atari DOS II Manual* so it will not be covered here.

OPEN — Open is not truly an extended operation, but for disk I/O we need to know that the DFM allows two additional “modes” beyond the fundamental OS modes (which are 4, 8, and 12 for read, write, and update). If ICAX1 contains a 6 when DFM is called for OPEN, then the disk DIRECTORY is opened (instead of a file) for read-only access. The filename now specifies the file (or files, if wild cards are used) to be listed as part of a directory listing. Note that DFM expects this type of OPEN to be followed by a succession of GETREC (get text line) OS calls (and we present an example of this below). If ICAX1 contains a 9, the specified file is opened as a write-only file, but the file pointer is set to the current end-of-file. Caution: DFM only appends on sector boundaries (normally this is transparent to the user, but *caveat artificer*).

Error Handling

This may not be the best place to introduce this topic, but the information is needed for examples which follow. Space doesn't permit a listing of all the I/O error codes, so we must refer you again to the BASIC and/or DOS II reference manuals. There are four fundamental kinds of errors that can occur with Atari OS:

HARDWARE ERRORS — Such as attempting to read a bad disk, write a read-only disk, etc.

SERIAL BUS ERRORS — Errors which occur when data is transferred between the computer and a peripheral device. Examples include Device Timeout, Device NAK, Framing Error, etc.

DEVICE DRIVER ERRORS — Found by the driver for the given device, as in (for the DFM) File Not Found, File Locked, Invalid Drive Number, etc.

OS ERRORS — Usually fundamental usage problems, such as Bad Channel Number, Bad Command, etc.

On return from any OS call, the Y-register contains the completion code of the requested operation. A code of one (1) indicates “normal status, everything is okay.” (I know, why not zero, which is easier to check for? Remember, I said Atari was good, not perfect.) By convention, codes from \$02 to \$7F (2 through 127 decimal) are presumed to be “warnings.” Those from \$80 to \$FF (128 through 255 decimal) are “hard” errors. These choices facilitate the following assembly language sequence:

```
JSR CIOV    ; call the OS
TYA         ; check completion code
BMI OOPS    ; if $80-$FF, it must be an error
```

In theory, Atari's OS always returns to the user with condition codes set such that the TYA is unnecessary. In practice, that's probably true; but a little paranoia is often conducive to longer life of both humans and their programs.

A Real, Live Example

Believe it or not, you now have all the information you need to do from assembly language any and all I/O done by Atari BASIC and/or BASIC A+ (excepting graphics, but that's coming...hold your breath). In an attempt to make you believe this statement, we will write a program in both BASIC and assembly language.

The BASIC Program

```
100 DIM BUFFER$(40)
200 OPEN #1,6,0,"D:*.**"
300 TRAP 700
400 INPUT #1,BUFFER$
500 PRINT BUFFER$
600 GOTO 400
700 CLOSE #1
```

This program will list all files on disk drive or (D1:) to the screen. This is exactly equivalent to using the “A” option of Atari's menu “DOS” (after then hitting RETURN for the filename) or to using “DIR” from OS/A+. Admittedly, this program easily improved. For example, replace line 200 with

```
200 INPUT BUFFER$: OPEN #1,6,0,BUFFER$
```

and now you can choose to list only some files. You might also wish to send the listing to the printer (change PRINT to LPRINT). However, we will leave such changes as an exercise to the reader and discuss only our simplified version.

Please now refer to the listing in Program 1. Since it follows the scheme of the above BASIC listing, it is almost self-explanatory. A few words are in order, though. The equates at the beginning have been kept to a minimum; I refer you to the “SHOOT” listing in **COMPUTE! #16** if you want a comprehensive list. (The mnemonics used are not all identical to those in the “SHOOT” listing; those shown are from our standard equates file.)

The program is intended to be called from BASIC via the USR function. However, no check is performed to see if the BASIC program were coded as (for example) PRINT USR(1600,0) instead of just PRINT USR(1600). (Note that 1600 decimal = 640 hex, the starting address.) If you would like to test this program with the BUG debug monitor, you should replace the RTS at the end of the program with a BRK before saying ‘G641’ (641 to avoid the PLA).

All errors, including an error on the OPEN DIRECTORY call to OS, are treated as end-of-file. A better program would verify the error status and print a message or some such. As an example of a

minor improvement, at LINE700 one could save the Y-register (status) value in FR0 and zero in FR0 + 1 (\$D4 and \$D5), thus returning the error code to the calling BASIC program.

Notice that values stored into the IOCB for FILE0 (the console screen output) were stored directly into ICCOM, etc., without an X-register offset. This is perfectly valid, so long as the X-register contains the proper value on calling CIO. In fact, we could have stored the values for FILE1 (the directory) by coding (for example) STA ICCOM + FILE1. Obviously, this technique only works when one uses a constant channel number; but most BASIC programs and many language programs can use predefined channel numbers.

There isn't really much more to say other than, "Try it!" It really does work. And, even if you don't understand the concepts on first reading, actually entering the program and following the program flow and remarks might give you a painless introduction to I/O from assembly language.

The Easiest Way Of Making Room?

With an ATARI 400 or 800, there are many ways and places to find "safe" hunks of memory, places to put assembly language routines, player/missile graphics, character sets, etc. Many of the programs that I have seen involved techniques that I consider risky. For example, moving BASIC's top of memory down requires that one do so only after issuing a GRAPHICS, command for the most memory-consuming graphics mode used in the program.

Other programs use machine language subroutines; but such subroutines must themselves have a place to stay. The best of such routines, however, approach the "official" Atari method. The approved method is normally used (by Atari) to add device drivers to the OS; in fact, the drivers for both DOS and the RS-232 ports follow these rules:

1. Inspect the system LOMEM pointers.
2. Load your routine (or reserve your buffer) at the current LOMEM.
3. Add the size of the memory you used to LOMEM and
4. Store the resultant value back into LOMEM.

If each routine, driver, etc., followed these rules, one could reserve more and more of memory without disturbing any following routine. (In fact, Atari drivers presume that LOMEM will never grow beyond 16K, \$4000, or even less; but the principle holds.) Actually, there's a hole in the above method: if the SYSTEM RESET button is pushed, OS goes through and resets all its tables, including the value in LOMEM. A "good" device driver can even take this into account, but we are going to make a few presumptions that are generally

valid.

By now, you should realize that all of BASIC's fundamental I/O commands are simply implementations of OS calls. PRINT becomes PUT TEXT RECORD; INPUT becomes GET TEXT RECORD. OPEN and CLOSE are essentially unchanged. In fact, the only BASIC commands that are not obvious clones of their assembly language counterparts are GET and PUT. Suffice it to say that these are actually simply special case implementations of GET BINARY RECORD and PUT BINARY RECORD (commands 7 and 11) where the buffer length is set to one byte.

Next month, we tackle the task of understanding how device drivers work, and we actually write a new and useful one that talks to a device built into *all* Atari machines (but one that Atari didn't provide a driver for). And we haven't forgotten the promise to show how graphics routines (such as PLOT and DRAWTO) are actually I/O routines.

The trick: BASIC always, repeat always, LOADs new programs at what it perceives LOMEM to be! Unfortunately, BASIC keeps its own MEMLOW pointer, which is loaded from LOMEM only on execution of a NEW, not on execution of LOAD or RUN and (significant!!!) not even in the case of SYSTEM reset. However, when there's a will...

— ATARI BASIC —

```
10 LOMEM = 743 : MEMLOW = 128
20 ADDR = PEEK(LOMEM) + 256 * PEEK
   (LOMEM + 1)
30 ADDR = ADDR + SIZE
40 HADDR = INT(ADDR / 256) : LADDR = ADDR
   - 256 * HADDR
50 POKE LOMEM, LADDR : POKE LOMEM + 1,
   HADDR
60 POKE MEMLOW, LADDR : POKE MEMLOW + 1,
   HADDR : RUN "D:PROGRAM2"
```

— BASIC A + —

```
10 lomem = 743 : memlow = 128
20 addr = dpeek(lomem) : dpoke lomem, addr + size
30 dpoke memlow, addr + size : run "D:PROGRAM2"
```

The above listing is Program A, whose only purpose in life is to set up memory for the real program, Program B. "SIZE" is the amount of memory to be reserved. The program changes both the system and BASIC bottom-of-usable-memory pointers so that either NEW or RUN "..." will recognize the reserved memory. The beginning lines of PROGRAMB follow:

— ATARI BASIC —

```
10 LOMEM = 743 : MEMLOW = 128
20 POKE LOMEM, PEEK(MEMLOW) : POKE
   LOMEM + 1, PEEK(MEMLOW + 1)
```

— BASIC A + —

```
10 dpoke 743, dpeek(128)
```


The only reason for these lines in PROGRAMB is the case of SYSTEM RESET. If the user types RUN after the reset, BASIC will copy its MEMLOW (the value which includes the reserved space!) into the system's LOMEM, just so they agree with each other. A caution: I don't know what will happen if you hit SYSTEM RESET as BASIC is in the process of loading PROGRAMB.

As far as I can tell, the only real problem that could occur would be if SYSTEM RESET were followed by a "DOS" command from BASIC. The DOS would then get control, thinking that LOMEM had not been changed. In a normal running program environment, though, this is, at worst, unlikely, so this method seems more than adequate.

Columnar Output

A problem inherent in Atari BASIC is that the default tabbing (when using 'PRINT exp,exp') is ten columns while the screen is 38 columns wide. This produces an output something like this:

```
PRINT 1,2,3,4,5,6,7,8,9,10
```

```
1      2      3      4
5      6      7      8
9      10
```

Not too pretty. POKE 82,0 will change the left margin of the screen to zero (default is column 2), producing a 40 column screen and thus making 10 column tabbing an excellent choice. Unfortunately, many TV sets have too much overscan to handle a true 40 column screen. Fortunately, Atari BASIC allows one to change the number of columns used in tabbing via a POKE 201, <tabwidth>. But

the only factors of 38 are 19 and 2, meaning you can have 19 columns of 2 characters each or 2 columns of 19 characters each. Not much improvement so far.

Consider, though, the table of factors shown in Figure 1. As an example, if we have a screen 36 characters wide, we can have 2,3,4,6,9,12, or 18 columns. And to get a screen 36 characters wide is easy: just POKE 83,37 (presuming that location 82 still contains a 2). So look at the list of factors, choose a screen width of N, and you can use a tabwidth equal to any factor. NOTE: a tabwidth of two will not print numerics in only two columns.

Finally, consider the flexibility available by judiciously choosing your tabwidth setting:

```
20 POKE 201,4 : PRINT 1,2,
30 POKE 201,7 : PRINT 3,
40 POKE 201,10 : PRINT 4,5
```

Printing various values in a loop with this method can actually produce some quite readable columnar listings.

N	Factors of N
40	2,4,5,8,10,20
39	3,13
38	2,19
37	none
36	2,3,4,6,9,12,18
35	5,7
34	2,17
33	3,11
32	2,4,8,16

Figure 1.

```
000      1000      .TITLE "DEMONSTRATION FOR DECEMBER COMPUTE"
```

```
EMONSTRATION FOR DECEMBER COMPUTE
SYSTEM EQUATES
```

```
000      1010      .PAGE "SYSTEM EQUATES"
      1020 ;
342      1030 ICCOM = $342 ; 'COMMAND', IN IOCB
344      1040 ICBADR = $344 ; 'BUFFER ADDRESS'
348      1050 ICBLN = $348 ; 'BUFFER LENGTH'
34A      1060 ICAUX1 = $34A ; 'AUX BYTE 1' (OPEN MODE)
      1070 ;
003      1080 COPN = 3 ; 'OPEN' COMMAND VALUE
005      1090 CGTXTR = 5 ; 'GET TEXT RECORD'
009      1100 CPTXTR = 9 ; 'PUT TEXT RECORD'
00C      1110 CCLOSE = 12 ; 'CLOSE'
      1120 ;
      1130 OPDIR = 6 ; 'OPEN DIRECTORY' SUB-COMMAND
      1140 ;
456      1150 CIO = $E456 ; WHERE TO CALL ATARI OS
      1160 ;
      1170 ; NOTE: OS/A+ users may omit lines 1010 thru 1160
```

```

1180 ; if they use .INCLUDE #D:SYSEQU.ASM
1190 ;
0000 1200 FILE0 = $00 ; IOCB NUMBER * 16
0010 1210 FILE1 = $10 ; IOCB NUMBER * 16
00FF 1220 LOW = $FF ; MASK FOR LSB OF ADDR
0100 1230 HIGH = $100 ; DIVISOR FOR MSE

```

DEMONSTRATION FOR DECEMBER COMPUTE
BEGIN ACTUAL PROGRAM

```

0000 1240 .PAGE "BEGIN ACTUAL PROGRAM"
1250 ;
1260 ; HOUSEKEEPING:
1270 ;
0000 1280 *= $640 ; PUT ALL THIS IN SAFE PLACE
0640 1290 .OPT OBJ ; WE DO WANT OBJECT CODE
1300 ;
1310 ; This program will list the
1320 ; directory of disk D1: to the
1330 ; E: device.
1340 ;
1350 ; Throughout, reference is made
1360 ; to the BASIC demo program
1370 ; which performs the same
1380 ; functions.
1390 ;
1400 DIR
1410 ; !!!! CAUTION !!!!
1420 ; If this routine is to be used
1430 ; from BASIC, the form MUST be
1440 ; xxx=USR(addr) as this routine
1450 ; makes no check on number of
1460 ; parameter bytes !!!
1470 ;
0640 68 1480 PLA ; PULL OFF # OF BYTES
0641 4C7206 1490 JMP START
1500 ;
1510 ; We jump around the buffer.
1520 ; Normally, the buffer would
1530 ; be at the end; but we simulate
1540 ; the BASIC program as closely
1550 ; as possible
1560 ;
1570 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
1580 ;
1590 ; 100 DIM BUFFER$(40)
1600 ;
0028 1610 BUFLen = 40
0644 1620 BUFFER *= *+BUFLen ; RESERVE 40 BYTES OF SPACE
1630 ;
1640 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
1650 ;
1660 ; 200 OPEN #1,6,0,"D:*.*)"
1670 ;
066C 44 1680 NAME .BYTE "D:*.*)",0
066D 3A
066E 2A
066F 2E
0670 2A
0671 00
1690 ; just a place to put filename

```

DEMONSTRATION FOR DECEMBER COMPUTE BEGIN ACTUAL PROGRAM

```

1700 ;
1710 START
1720 ; begin actual program
1730 ;
0672 A210 1740 LDX #FILE1
0674 A903 1750 LDA #COFN ; THE OPEN COMMAND
0676 9D4203 1760 STA ICCOM,X ; IS SET UP
0679 A906 1770 LDA #OPDIR ; MODE 6, DIR OPEN
067E 9D4A03 1780 STA ICAUX1,X ; THUS THE MODE
067E A96C 1790 LDA #NAME&LOW
0680 9D4403 1800 STA ICBADR,X ; LSB OF ADDR
0683 A906 1810 LDA #NAME/HIGH ; AND MSB OF ADDR
0685 9D4503 1820 STA ICBADR+1,X ; ...OF FLNM
0688 2056E4 1830 JSR CIO ; CALL ATARI OS
068E 98 1840 TYA ; CHECK STATUS
068C 3035 1850 BMI LINE700 ; HUH??
1860 ;
1870 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
1880 ;
1890 ; 300 TRAP 700
1900 ; SEE THE 'BMI' JUST ABOVE
1910 ;
1920 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
1930 ;
1940 ; 400 INPUT #1,BUFFER$
1950 ;
1960 LINE400
068E A210 1970 LDX #FILE1
0690 A905 1980 LDA #CGTXTR
0692 9D4203 1990 STA ICCOM,X ; 'INPUT' A LINE
0695 A944 2000 LDA #BUFFER&LOW
0697 9D4403 2010 STA ICBADR,X ; LSB OF ADDR
069A 8D4403 2020 STA ICBADR ; OF WHERE LINE GOES
069D A906 2030 LDA #BUFFER/HIGH
069F 9D4503 2040 STA ICBADR+1,X ; AND MSB
06A2 8D4503 2050 STA ICBADR+1 ; (WE ALSO SET UP ADDR FOR FILE #0)
06A5 A928 2060 LDA #BUFLEN
06A7 9D4803 2070 STA ICLEN,X ; BUFFER LEN
06AA 8D4803 2080 STA ICLEN ; IS MAX WE USE
06AD 2056E4 2090 JSR CIO ; AND GO GET A LINE
06B0 98 2100 TYA
06B1 3010 2110 BMI LINE700 ; "TRAP 700"
2120 ;
2130 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
2140 ;
2150 ; 500 PRINT BUFFER$
2160 ;
2170 ; note that PRINT automatically
2180 ; uses file #0, so we will do
2190 ; so also !!
2200 ;
2210 ; also note that we saved a few

```

DEMONSTRATION FOR DECEMBER COMPUTE BEGIN ACTUAL PROGRAM

```

2220 ; bytes by setting up the buffer
2230 ; address and length in 'LINE400'
2240 ;

```



```

B3 A909 2250 LDA #CPTXTR
B3 04203 2260 STA ICCOM ; PUT A LINE IS CMD
B8 A200 2270 LDX #FILE0 ; THE CONSOLE IS #0
BA 2056E4 2280 JSR CIO ; TO THE I/O
BD 98 2290 TYA
BE 3003 2300 BMI LINE700 ; OOPS?? HOW ???
2310 ;
2320 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
2330 ;
2340 ; 600 GOTO 400
2350 ;
C0 4C8E06 2360 JMP LINE400 ; SELF EXPLANATORY
2370 ;
2380 ;
2390 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
2400 ;
2410 ; 700 CLOSE #1
2420 ;
2430 LINE700
C3 A210 2440 LDX #FILE1
C5 A90C 2450 LDA #CCLOSE
C7 9D4203 2460 STA ICCOM,X ; COMMAND IS 'CLOSE'
CA 2056E4 2470 JSR CIO ; GO CLOSE THE FILE
CD 60 2480 RTS ; END OF ROUTINE
2490 ;
2500 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
2510 ;
CE 2520 .END

```

ATARI*800*OWNERS

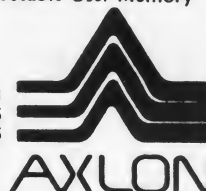


PLUG IN AND GO !

The Axlon RAMDISK Memory System provides 128K of RAM memory which can be utilized as an additional disk device or bank selectable RAM memory! The DOS supplied with the system allows you to utilize the RAMDISK Memory System as a disk device in conjunction with your Atari 810*. The system is up to 20 times faster than the Atari 810 and is compatible with existing Atari 800 software. As user memory, the RAMDISK Memory System is organized as eight (8) 16K banks. The system is installed with two 16K RAM modules giving you a 160K Atari 800 system. Drop by your local computer store for a demonstration or contact Axlon Inc. for more information.

- Plug-in Compatibility
- 128K Bytes of RAM Memory
- Compatible with existing Atari 800 Software
- Can be utilized as an additional disk - function for function, up to 20 times faster than the Atari 800
- Includes DOS Memory Management Software
- Can be utilized as Bank Selectable User Memory
- Gold Plated Contacts
- 90 Day Warranty

170 N. Wolfe Road
Sunnyvale, CA 94086
(408) 730-0216



* Indicates Trademark of Atari, Inc.

INSIGHT: ATARI

Bill Wilkinson
Cupertino, CA

I have recently seen a copy of the complete *De Re Atari* (by Atari's own Chris Crawford, author of *SCRAM* and *EASTERN FRONT*, et al). Since two out of three people I talk to say "Huh?" when I mention the name, I have personally subtitled it *Everything You Ever Wanted to Know About the Atari Computers But Didn't Know Enough to Ask*. The book concerns itself with foibles, tricks, innards, hardware, software, and everything in between: there are even tricks using Atari BASIC (that are "obvious" upon discovery) which we never thought about when we designed the thing! I must heartily recommend that every serious Atari programmer trade in his or her left thumb, if necessary, for a copy of this book.

"De Re" (the insiders' appellation) is currently being serialized in *BYTE* magazine (I guess Atari's trying to impress the non-Atari world), but seeing the book in one piece is somehow more instructive. "De Re" is generally a fantastic resource, but it does often assume that the reader has intimate knowledge and understanding of the Atari Hardware Reference Manuals, etc. This is *not* a fault (the authors forewarn the reader); and, besides, it does leave room for columns like this. I don't intend to duplicate material in either Atari's manuals or "De Re", but there is bound to be some overlap. I intend to present the "hows" and "whys" to supplement Atari's "whats."

I try to write this column for the programmer: the person who knows software, but is unfamiliar with Atari hardware and/or Atari's system level software. If this column stretches your understanding of the Atari and/or its software, that's probably good. And I am constantly amazed at the questions which beginners on the Atari come up with; they often show "insights" to solution methods that I wouldn't dream of. The first questions are arriving in my mailbox. Send more!

This month's column is part three of the series on the Atari Operating System. Next month we will cover screen output, including graphics, to formally end the series. I have a few ideas on what should come next for you non-BASIC Atari users, but I would welcome some input. Also, this month, we begin a series which will explore the inner workings of Atari BASIC.

Atari I/O, Part 3: Device Handlers

As we noted before, Atari's OS is actually a very

small program (approximately 700 bytes). Even so, it is able to handle the wide variety of I/O requests detailed in the first two parts of this series with a surprisingly simple and consistent assembly language interface. Perhaps even more amazing is the purity and simplicity of the OS interface to its device handlers.

Admittedly, because of this very simplicity, Atari's OS is sometimes slower than one would wish (probably only noticeably so with *PUT BINARY RECORD* and *GET BINARY RECORD*) and the handlers must be relatively sophisticated. But not overly so, as we will show.

The Device Handler Table

Atari OS has, in ROM, a list of the standard devices (P:,C:,E:,S:, and K:) and the addresses thereof. So far, so good. But notice that, for example, the disk handler (D:) is not listed there; how does OS know about other devices? Simple. On *SYSTEM RESET*, the list is moved from ROM to RAM, and OS then utilizes only the RAM version. To add a device, simply tack it on to the end of the list: you need only specify the device's name (one character) and the address of its handler table (more on that in a moment). To reassure you that it is this simple, let me point out that this is exactly how the "D:" (Disk) handler is attached when the disk is booted.

In theory, all named device handlers under Atari OS may handle more than one physical device. Just as the disk handler understands "D1:" and "D2:", so could a printer handler understand "P1:" and "P2:". In practice, of all the standard Atari handlers only the Disk and Serial Port handlers can utilize the sub-device numbers. Incidentally, Atari OS supplies a default sub-device number of "1" if no number is given (thus "D:" becomes "D1:"). A project for those of you with two printers (there

*=	\$031A	
HTABS		
.WORD	PDEVICE	; the Printer device ; and the address of its driver
.BYTE	'C'	; the Cassette device
.WORD	CDEVICE	
.BYTE	'E'	; the screen Editor device
.WORD	EDEVICE	
.BYTE	'S'	; the graphics Screen device
.WORD	SDEVICE	
.BYTE	'K'	; the Keyboard device
.WORD	KDEVICE	
.BYTE	0	; zero marks the end of the table
.WORD	0	; ...but there's room for up to
.BYTE	0	; ...9 more devices
	et cetera	

Figure 1.

must be one or two of you): presumably one of them is connected via the MacroTronics interface; if so, try modifying the MacroTronics handler so that "P1:" refers to the Atari 850 interface while "P2:" refers to the MacroTronics. It's really a fairly easy project, presuming you have the listings of Atari's OS (which are available from Atari).

Rules For Writing Device Handlers

Each device which has its handler address placed into the handler address table (above) is expected to conform to certain rules. In particular, the driver is expected to provide six action subroutines and an initialization routine. (In practice, I believe the current Atari OS only calls the initialization routines for its own pre-defined devices. Since this may change in future OS's and since one can force the call to one's own initialization routine, I must recommend that each driver include one, even if it does nothing.) The address placed in the handler address table must point to, again, another table, the form of which is shown in Figure 2.

Notice the six addresses which must be specified; and note that, in the table, one must subtract one from each address (the "-1" simply makes CIO's job easier...honest). A brief word about each routine is in order.

The OPEN routine must perform any initialization needed by the device. For many devices, such as a printer, this may consist of simply checking the device status to insure that it is actually present. Since the X-register, on entry to each of these routines, contains the IOCB number being used for this call, the driver may examine ICAX1 (via LDA ICAX1,X) and/or ICAX2 to determine the kind of OPEN being requested. (Caution: Atari OS preempts bits 2 and 3, \$04 and \$08, of ICAX1 for read/write access control. These bits may be examined, but should normally not be changed.)

The CLOSE routine is often even simpler. It should "turn off" the device if necessary and if possible.

The PUTBYTE and GETBYTE routines are just what are implied by their names: the device handler must supply a routine to output one byte to the device and a routine to input one byte from the device. *However*, for many devices, one or the other of these routines doesn't make sense (ever tried to input from a printer?). In this case the routine may simply RTS and Atari OS will supply an error code.

The STATUS routine is intended to implement a dynamic status check. Generally, if dynamic checking is not desirable or feasible, the routine may simply return the status value it finds in the user's IOCB. However, it is *not* an error under Atari OS to call the status routine for an unOPENed

device, so be careful.

The XIO routine does just what its name implies: it allows the user to call any and all special and wonderful routines that a given device handle may choose to implement. OS does nothing to process an XIO call except pass it to the appropriate driver.

Note: In general, the AUXilliary bytes of each IOCB are available to each driver. In practice, it is best to avoid ICAX1 and ICAX2, as several BASIC and OS commands will alter them at their will. Note that ICAX3 through ICAX5 may be used to pass and receive information to and from BASIC via the NOTE and POINT commands (which are actually special XIO commands). Finally, drivers should not touch any other bytes in the IOCBs, especially the first two bytes.

Notice that handlers need not be concerned with PUT BINARY RECORD, GET TEXT RECORD, etc.: OS performs all the needed house keeping for these user-level commands.

HANDLER

.WORD	<address of OPEN routine>-1
.WORD	<address of CLOSE routine>-1
.WORD	<address of GETBYTE routine>-1
.WORD	<address of PUTBYTE routine>-1
.WORD	<address of STATUS routine>-1
.WORD	<address of XIO routine>-1
JMP	<address of initialization routine>

Figure 2.

Rules For Adding Things To OS

We touched on this subject last month, in the section titled "The Easiest Way of Making Room?", but a review and an addition are in order. Both Atari FMS (File Manager System, otherwise known as DOS and/or the Disk Device Driver) and the serial port handlers follow the same scheme when they add themselves to OS, so it is safe to assume that this method may be considered the *de facto* Atari standard. We enumerate:

1. Inspect the system MEMLO pointer (at \$2E7, I called it LOMEM last month, which is BASIC's name for it).
2. Load your routine (including needed buffers) at the current value of MEMLO.
3. Add the size of your routine to MEMLO.
4. Store the resultant value back in MEMLO.
5. Connect your driver to OS by adding its name and address into the handler address table.
6. Fool OS so that if SYSTEM RESET is hit steps 3 through 5 will be re-executed (because SYSTEM RESET indeed resets the handler

address table and the value of MEMLO).

In point of fact, step 2 is the hardest of these to accomplish. In order to load your routine at wherever MEMLO may be pointing, you need a relocatable (or self-relocatable) routine. Since there is currently no assembler for the Atari which produces relocatable code, this is not an easy task. However, I just happen to have a method which works. But it will have to wait for a later article.)

Step 6 is accomplished by making Atari OS think that your driver is the Disk driver for initialization purposes (by "stealing" the DOSINI vector) and then calling the Disk's initializer yourself when steps 3 through 5 are performed. This is a fairly simple process, but again, details must await a future article.

Yet Another Real Live Example

I promised last month that we would present a driver for a "peripheral" device found in every Atari, yet not supported by any Atari device handlers. I could have been cagey and presented a driver for a "Null" device. (A handy thing to have, actually: One can throw away one's output *very* fast when trying to debug a program. See *De Re Atari* for a simple implementation of one. Better yet, try to write one from the information presented herein.) Being a glutton for punishment, I undertook to write a truly useful handler for Atari's overlooked device: RAM memory!!

After the snickers and sarcastic comments die down, let me point out how truly useful such a device is to BASIC programs: program one can "write" data to RAM and then chain to program two, which then "reads" the same data back. Voila! Chaining with COMMON in Atari BASIC. So herewith the "M:" (Memory) driver, presented in its entirety in Figure 3.

Does It Work?

Some words of caution are in order. This driver does *not* perform step 6 as noted in the last section (but it may be reinitialized via a BASIC USR call). It does *not* perform self-relocation: instead it simply locates itself above all normal low memory usage (except the serial port drivers, which would have to be loaded *after* this driver). If you assemble it yourself, you could do so at the MEMLO you find in your normal system configuration (or you could improve it to be self-modifying, of course).

Other caveats pertain to the handler's usage: it uses RAM from the contents of MEMTOP (\$2E5) downward. It does *not* check to see if it has bumped into BASIC's MEMTOP (\$90) and hence could conceivably wipe out programs and/or data. To be safe, don't write more data to the RAM than a FRE(0) shows (and preferably even less).

In operation, the M: driver reinitializes upon an OPEN for write access (mode 8). A CLOSE followed by a subsequent READ access will allow the data to be read in the order it was written. More cautions: don't change graphics modes between writing and reading if the change would use more memory (to be safe, simply don't change at all). The M: will perform almost exactly as if it were a cassette file, so the user program should be data sensitive if necessary: the M: driver will *not* itself give an error based on data contents. Note that the data may be re-READ if desired (via CLOSE and re-OPEN).

Installing The M: Driver

The most obvious way to install this driver (Program 1) is to type in the source and assemble it directly to the disk. Then simply loading the object file from DOS 2 (or OS/A+) will activate the driver and move LOMEM as needed. You could even name the resulting file "AUTORUN.SYS" so that it would be automatically booted on power up.

If you don't have an assembler and/or disk, the problem is a little more difficult. If you are comfortable writing BASIC programs that load assembly language data to memory, you might use the techniques described in last month's "Make Room?" to reserve the required memory. Then a simple POKER program which uses DATA statements would suffice.

But the assembly listing given here is designed for a disk system and would waste 5K bytes or so in a cassette system. So, if you can't reassemble it and/or write that POKER program, you will just have to be patient: I will try to give you a simplified BASIC POKER program next month.

A suggested set of BASIC programs is presented:

Ending of Program 1:

```
9900 OPEN #2,8,0,"M:"
9910 PRINT #2; LEN(A$)
9920 PRINT #2; A$
9930 CLOSE #2
9940 RUN "D:PROGRAM2"
```

Beginning of Program 2:

```
100 JUNK = USR(7984)
    [ to insure the M: driver is linked, in case of
      RESET ]
110 OPEN #4,4,0,"M:"
120 INPUT #4, SIZE
130 DIM STRING$(SIZE)
140 INPUT #4, STRING$
150 CLOSE #4
```

BASIC A+ users might find RPUT/RGET and BPUT/BGET to be useful tools here instead of PRINT and INPUT. And, of course, users of any other language(s) might find this a handy inter-program communications device.

BASIC, Part 1: Why?

The first "Why?" I usually hear is "Why not Microsoft BASIC?" After a little probing, I find that the question really boils down to "Why not string arrays?" There is no simple answer to that question, so I hope to save myself time in the future by pointing toward these articles. Because I intend to give the true and not-so-simple answer, along with some (hopefully) very interesting information.

Believe it or not, Atari BASIC pretty much works the way it was designed and specified. And yours truly must take a large part of the brickbats or roses you might throw because of those specifications. We (that is, at the time, Shepardson Microsystems) were just finishing the highly successful and very powerful Cromemco 32K Structured BASIC. And, while a few Cromemco users had carped about the lack of string arrays, on the whole the real power of the language is extraordinarily impressive. All this "power" probably went to our head(s), so of course we had to duplicate the feat for Atari.

Oops. A small problem: Cromemco gave us 32K bytes for Structured BASIC; Atari gave us 10K bytes. What comes out? Wrong question! What can stay in?! Of course, Atari had some ideas, too, and the important features that we ended up with include (in my opinion):

Decimal Arithmetic
Long Variable Names
Long Strings (more than 255 bytes)
Flexible I/O
Reasonable Assembly
Language Interface
Syntax Check at entry time

That last item won't be appreciated by those of you who haven't used a BASIC that doesn't do it, so I will try to describe the horrors to you: You type in a long program which includes a line such as:



CRYPTS OF TERROR

Beware as you enter the Crypts Of Terror. No one has survived this horror. Only your unrelenting nerve and determination will drive you deeper into the unknown.

Find what lurks in these ancient crypts!!

At last we have found an adventure with full graphics, sound and intrigue for your ATARI 400/800 computer.

• CRYPTS OF TERROR is the first adventure game that was completely designed for the Atari computers only. The graphics are the finest available using the full potential of the Atari.



Atari 800/400 16K requires joysticks.

Payment: Personal Checks – allow three weeks for check to clear.

American Express, VISA, MasterCard – include all numbers on card. Please include phone number with all orders.

Orders from USA \$29.95 (US funds)

Orders from Canada \$39.95 (Canadian funds)

Plus \$2.00 for shipping.

Ontario residents add 7% R.S.T.

Check your local computer dealer for Crypts Of Terror.

Dealer inquiries encouraged.

**INHOME
SOFTWARE**

PH. 1-416-961-2760

1560 Yonge St.
P.O. Box 10
Toronto
Ontario Canada
M4T 1Z7

FREE SHIPPING FREE SHIPPING FREE SHIPPING

ATARI CENTRAL GOES NATIONAL!!!

**HW ELECTRONICS—THE #1 SOURCE FOR
QUALITY ATARI PRODUCTS!**

ATARI 400/800 TECHNICAL USER NOTES

A MUST for anyone wishing to delve into the powers of the ATARI computer system. Includes detailed information of the hardware (including schematics) as well as the Operating System.

Cat No. 3141

\$27.00

AN INVITATION TO PROGRAMMING 2

Learn how to write programs in BASIC. These lessons cover library functions, FOR...NEXT loops, subroutines, and READ, DATA, DIMension, PEEK and POKE statements. They also cover flow charting and programming structure. Includes two cassettes and a workbook.

Cat No. 3250 8K, Cass.

\$24.95

AN INVITATION TO PROGRAMMING 3: SOUND & GRAPHICS

The sound cassette covers simple music theory and ATARI BASIC commands for setting the sound registers to the desired pitch, purity, and loudness levels. The Graphics cassette teaches you to use the color registers, the graphics characters, the SET-COLOR and POSITION statements, and graphics modes 0 through 5. Includes two cassettes and a workbook.

Cat No. 3251 8K, cass.

\$24.95

LE STICK

by DATASOFT

The joystick of the future. Internal motion detectors maneuver your sights in any direction with simple one handed movements. The large pushbutton provides a quick response to your firing commands.

Cat No. 2925

\$39.95

EASTERN FRONT

EASTERN FRONT simulates Operation Barbarossa, the German invasion of Russia during World War II. The use of intricate artificial intelligence routines and high-resolution, smooth-scrolling terrain maps eliminate the usual drudgery of playing wargames. To top it off, multiprocessing permits simultaneous moves by both you and the computer.

Cat No. 3294 16K, cass.

\$26.95

Cat No. 3295 32K, disk

\$29.95

HOW TO ORDER

Mention this ad and WE PAY SHIPPING (UPS ground-USA). Call or write. Pay by check, M/C, Visa, or COD (add \$1.40 for COD).

Offer expires Feb. 1, 1982.

HW ELECTRONICS

19511 Business Center Dr. Dept. G1

Northridge, CA 91324

(800) 423-5387 (213) 886-9200

**3034 IF SYSTEMERROR THEN
PINT "Bad Disk Drive":
GOTO 4090**

Did you catch it? It says 'PINT' where it should say 'PRINT'.

Most microcomputer BASICs will happily gulp that line in with nary a burp. Now, 13 months later, when that dreaded 'systemerror' actually occurs, your user (who lives in Hong Kong, naturally) sees the helpful message

*****SYNTAX ERROR at LINE 3037**

When you have fathomed the implications of that, calm your nerves so we can continue.

Needless to say, we were more than happy to include the Syntax Check feature. However, this inclusion had implications that rippled throughout the rest of the design of BASIC. First, you don't get something for nothing: such syntax checking uses memory, perhaps one to two kilobytes. Second, pre-syntaxing implies that the user program will be "tokenized": that is, the user's source will be converted into internal tokens for ease of execution and efficiency. Even Microsoft BASICs tokenize the keywords of the language; Atari BASIC tokenizes *everything*: keywords, variables, constants, operators, etc. Thirdly, the decision to have strings longer than 255 characters (coupled with the tight memory requirements) simply precluded any implementation of string arrays. (In fact, I do not know of *any* small-machine BASIC that supports string arrays with elements longer than 255 characters.)

Before perusing some quickie programs to show the effects of tokenizing, I should like to give some credit where it is due.

Though I participated in the specifications for Atari BASIC, I had little to do with the actual implementation. More history: Atari asked us (in September, 1978) to bid on producing a custom "consumer-oriented" BASIC

for them. Sometime in October, the specifications were finalized and Paul Laughton and Kathleen O'Brien (with a very little help from three more of us) began to work in earnest. The contract called for delivery by April 6, 1979, and included delivery of a File Manager System (DOS 1). Atari planned to take an early, 8K Microsoft BASIC to the Consumer Electronics Show (in Las Vegas) in January, 1979, and then switch later. The actual purchase order took a while to get through Atari's red tape, and the final version thereof is dated 12/28/78 — about one week *after* both BASIC and DOS were delivered to Atari! Atari took Atari BASIC to CES.

Investigating BASIC's Tokens

There are three fundamental types of tokens in Atari BASIC, each of which occupies exactly one byte of RAM memory, with only two special cases. The token types are statement name tokens, operator name tokens (which include function names and some other miscellany), and variable name tokens. The special cases are numeric and string constants, which begin with an operator name token, but are followed by the actual value of the constant.

Statement name tokens can *only* occur as the first item of a statement and, thus, have their own keyword and tokenizing table. In theory, Atari BASIC's structure could support up to 256 types of statements. Variable name tokens and operator name tokens are intermixed throughout the rest of a statement and are distinguished by the state of their upper bit: variable name tokens have their upper bit on, operators don't.

A few of the statement types are also special cased in that they are not followed by operator and variable tokens. These special cases include the

**ALL ATARI® HARDWARE 15%-25%
OFF LIST PRICE**

	OUR PRICE	SAVE
Atari 400 w/16K	\$320	20%
Atari 800 w/16K	\$810	25%
Atari 410 cassette	\$ 67	25%
Atari 810 disk drive	\$480	20%

**ATARI® ACCESSORIES 10%-20%
OFF LIST PRICE**

8K Memory Board	\$45	10%
16K Memory Board	\$80	20%
Joysticks (pair)	\$17	15%
Paddles (pair)	\$17	15%

To order: Call 617-964-3080
Ask for mail order, or write

The Bit Bucket
1355 Washington Street (Rt. 16)
West Newton, MA 02465
617-964-3080

**PLUS 10%-20% OFF
ALL ATARI® SOFTWARE
ALSO 3RD PARTY HARDWARE
AND SOFTWARE AT
COMPARABLE SAVINGS**

A Revolutionary Concept In Software
For The ATARI® 400 and 800 Computers

The Interactive Storybook

Sammy The Sea Serpent

A Storybook Program For Children Ages 4 to 7.

Sammy The Sea Serpent

is the story of an imaginary sea creature who is lost and trying to find his way home. The story is read aloud to your child by a professional actress. While the tale is being told, the child uses the joystick to help Sammy out of some tight spots.

The A side of the cassette contains the interactive story; the B side contains games that the child plays with Sammy.

The program uses voice, sound effects; music, color and mixed graphics.

Sammy The Sea Serpent

can be used with either the ATARI 400 or 800 and requires 16K. It is available in cassette format only. Price is \$16.95 plus \$2.00 shipping and handling.

Also available at fine computer stores.



Program Design, Inc./11 Idar Court Greenwich, CT 06830
203-661-8799

ATARI is the registered trademark of ATARI, Inc.

ATARI 800 SOFTWARE!!

**TRS-80 EXTENDED BASIC
COLOR COMPUTER SOFTWARE**

**TRS-80 POCKET COMPUTER
SOFTWARE**

FROM: SEBREE'S COMPUTING
456 Granite Avenue, Monrovia, Calif., 91016

ATARI 800 16k min. 40K Preferred. >>>GRAPHICS EDITOR!!<<< NOW, both 2-D and 3-D scenes can be designed with a JOYSTICK, and then saved to disk!! These scenes can then be loaded in later to be edited before you save it again under another name!! All of this can be done using ANY graphics mode!! But that's not all!! You can save entire screens OR just individual images in 2-D OR 3-D!! You have the option of giving the new scenes different file names, a MENU of disk files is shown on the screen as you choose the file name. A 'HELP' option is included should you have trouble with ANY operation. If you decide to use the 3-D option, you may change the 3-D view(s) of the object(s) on the screen. Uses Player-Missile Graphics for the 'cursor'. POWERFUL, AND WELL DOCUMENTED!! Disk version is recommended (--add \$5.00--). W/5 Programs. ONLY \$29.95 +\$1.50 p/h. on Cassette.

COLOR COMPUTER EXTENDED BASIC>>>3-D C.C. GRAPHICS PACKAGE!!<<< NOW, you can get our popular 3-D Graphics Package for your Color Computer!! Design your own graphics with a Joystick and view these images from any angle you want!! Software selectable screen resolutions, Colors, Viewing angles, Rotation, object erase & replacement, wide-angle or telephoto views, along with ALL of the required 3-D operations to change viewers location.W/1 listings! ONLY \$24.95+\$1.50 p/h.cassette

>FLIGHT SIMULATOR<Req. 24K ATARI or 16K TRS-80 COL.CMP.A Graphic Flight simulator for 1 player. You have to take-off & navigate to the next airport -watching obstacles!! Then attempt to land at the airport- if you have enough fuel! Great graphics! Different difficulty levels.Requires one joystick.Only-\$17.95 +.95p/h

>TRIP TO JUPITER SPACE ADVENTURE!< Req. 24K ATARI or 16K COLOR COMPUTER. Launch your space craft from Earth & get on a trajectory to JUPITER! Obstacles to navigate through! Land on JUPITER and re-launch your craft & bring it to orbit and re-connect with the Mothercraft and head back to Earth!! ONLY-\$18.95 +.95 p/h

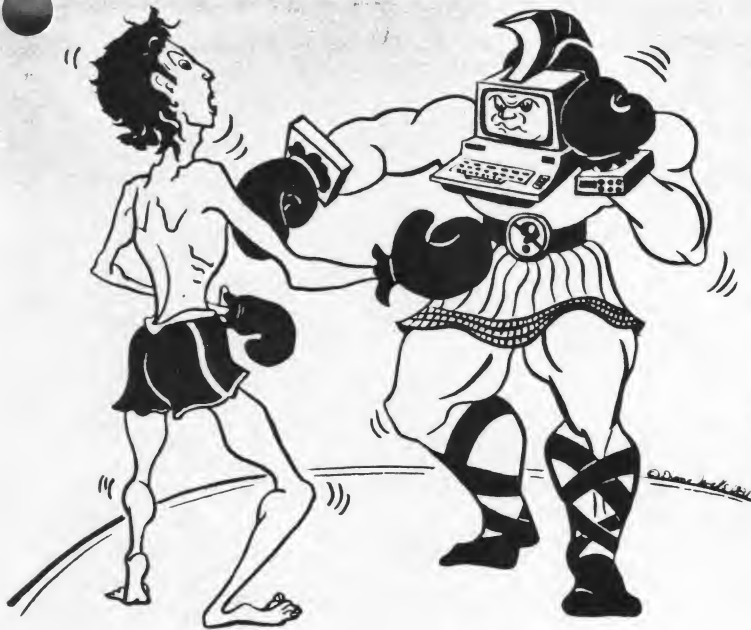
>3-D RED BARON DOGFIGHT/ FLIGHT SIMULATOR!< Req.16K ATARI or 16K C.CMP. NOW you can play this exciting 3-D simulation /game on both computers! Done in HI-RES!! You are in pursuit of the famed RED BARON, and catching up to him. If you don't shoot him down soon, his tail gunner starts shooting at you!! Out-of-the cockpit view, w/ALTIMETER, RADAR, BANKING METER, & NUMBER OF WINS. Only-\$16.95 +.95 p/h

TRS-80 POCKET COMPUTER>>WUMPUS ADVENTURE!<< Now you can play our popular WUMPUS game on your P.C. With A 4 page manual and listing, on cassette--\$7.95 +.95 p/h Pay by Check or Money Order (preferred).Foreign-U.S.funds ONLY.CAL.6% sales.tax

DON'T ASK PRESENTS:

ABUSE

For the ATARI 400/800



Match wits with your computer in an insult war! With **ABUSE** your computer becomes a slightly demented, smart-aleck insult-exchanger.

- **Millions** of different insults.
- Understands and responds to user input.
- Hybrid BASIC/machine language program.
- Game feature: become a Master of **ABUSE**.
- **Many** surprises to discover.

Release your aggressions! Inflict **ABUSE** on anyone who's got it coming!

REQUIRES 40K RAM AND BASIC CARTRIDGE

Dealer inquiries welcome

Available soon for the Apple II

At your computer store or send **\$19.95** + \$2.00 handling to:

DON'T ASK - 2265 Westwood Blvd. B-150 - Los Angeles, CA 90064 - (213) 397-8811

Calif. residents add 6% sales tax.

ATARI is a trademark of ATARI INC.
APPLE is a trademark of APPLE COMPUTER INC.

vious REM and DATA and the not-so-obvious ROR (the statement name given to lines containing a syntax error).

Since each variable is reduced to a single byte (with its upper bit set), there are a maximum of 128 different variable names per program. There is the further implication that BASIC must remember the association of name to token in order to LIST your program back to you. The actual ASCII names are stored in the "Variable Name Table," and we investigated its structure in **COMPUTE!** #17 under the heading of "VARIABLE,

VARIABLE, VARIABLE." (Briefly, the names are simply stored one after the other, with the upper bit of the last character of each name turned on.)

The statement and operator names are obviously predefined in the BASIC ROM cartridge, and we offer herewith a program (Program 2) which prints out the token numbers and corresponding keywords. When you run the program, you will notice that some operators (especially the left parenthesis) appear to be repeated. They are. We will find out why next month.

Program 1.

Sample device driver for Atari's OS
- general remarks ---

```

10      1000      .PAGE  "---- general remarks ----"
1010    ;;;;;;;;;;;;;;
1020    ;
1030    ; The "M:" driver --
1040    ; Using memory as a device
1050    ;
1060    ; Includes installation program
1070    ;
1080    ; Written by Bill Wilkinson
1090    ; for January, 1982, COMPUTE!
1100    ;
1110    ;;;;;;;;;;;;;;

```



```

1120 ;
1130 ; EQUATES INTO ATARI'S OS, ETC.
1140 ;
034A 1150 ICAUX1 = $34A ; The AUX1 byte of IOCB
1160 ;
0008 1170 OPOUT = 8 ; Mode 8 is OPEN for OUTPUT
1180 ;
02E7 1190 MEMLO = $2E7 ; pointer to bottom of free RAM
02E5 1200 MEMTOP = $2E5 ; pointer to top of free RAM
1210 ;
00E0 1220 FR1 = $E0 ; Fltg Pt Register 1, scratch
1230 ;
0001 1240 STATUSOK = 1 ; I/O was good
0088 1250 STATUSEOF = $88 ; reached an end-of-file
1260 ;
031A 1270 HATABS = $31A
1280 ;
0100 1290 HIGH = $100 ; divisor for high byte
00FF 1300 LOW = $FF ; mask for low byte
1310 ;

```

A sample device driver for Atari's OS
The installation routine

```

0000 1320 .PAGE "The installation routine"
1330 ;
0000 1340 *= $1F00
1350 ; This first routine is simply
1360 ; used to connect the driver
1370 ; to Atari's handler address
1380 ; table.
1390 ;
1400 LOADANDGO
1F00 A200 1410 LDX #0 ; We begin at start of table
1420 SEARCHING
1F02 BD1A03 1430 LDA HATABS,X ; Check device name
1F05 F00A 1440 BEQ EMPTYFOUND ; Found last one
1F07 C94D 1450 CMP #'M ; Already have M: ?
1F09 F01A 1460 BEQ MINSTALLED ; Yes, don't reinstall
1F0B E8 1470 INX
1F0C E8 1480 INX
1F0D E8 1490 INX ; Point to next entry
1F0E D0F2 1500 BNE SEARCHING ; and keep looking
1F10 60 1510 RTS ; Huh? Impossible!!!
1520 ;
1530 ; We found the current end of the
1540 ; table...so extend it.
1550 ;
1560 EMPTYFOUND
1F11 A94D 1570 LDA #'M ; Our device name, "M:"
1F13 9D1A03 1580 STA HATABS,X ; is first byte of entry
1F16 A93B 1590 LDA #MDRIVER&LOW
1F18 9D1B03 1600 STA HATABS+1,X ; LSB of driver addr
1F1B A91F 1610 LDA #MDRIVER/HIGH
1F1D 9D1C03 1620 STA HATABS+2,X ; and MSB of addr
1F20 A900 1630 LDA #0
1F22 9D1D03 1640 STA HATABS+3,X ; A new end for the table
1650 ;
1660 ; now change LOMEM so BASIC won't
1670 ; overwrite us.

```

Imagine being able to print the letter "A" and get a multi-color space ship. Using THE NEXT STEP and a minimum of programming effort, you can do it in no time at all.

THE NEXT STEP contains well-written, easy-to-use documentation with simple BASIC programming examples that show you how THE NEXT STEP can help develop colorful graphic displays. Graphics you never thought possible until now.

THE NEXT STEP is a user friendly, menu driven graphics tool kit that allows you to create new character sets or redefine characters to make shapes for use with your basic or machine language programs. THE NEXT STEP allows you to save these "new" characters on disk for future use.

THE NEXT STEP is perfect for use on shapes for animation and features a joystick controlled color menu to make your graphics come alive. THE NEXT STEP even generates its own code to help you incorporate new characters and shapes into your programs.

THE NEXT STEP allows you to see your shapes as you make them. Now you can determine ahead of time how characters will interact with one another when creating shapes for Character Set or Player-Missile Graphics.

THE NEXT STEP helps you to mix any of ATARI's 14 graphics modes in the same display. THE NEXT STEP is a perfect graphics utility for the BASIC or machine language programmer- novice and professional alike.

THE NEXT STEP features full joystick control for ease-of-use and quick editing.

THE NEXT STEP runs on any 32K ATARI 400/800 with a disk drive and is available for \$39.95 at your local computer store or order direct from



THE NEXT STEP

VISA, MASTERCARD, CHECK, C.O.D.

Add \$1.00 for Shipping

36575 MUDGE RANCH ROAD • COARSEGOLD, CA 93614 • 209-683-6858

```

1680 ;
1690 MINSTALLED
F25 A900 1700 LDA #DRIVERTOP&LOW
F27 8DE702 1710 STA MEMLO ; LSB of top addr
F2A A920 1720 LDA #DRIVERTOP/HIGH
F2C 8DE802 1730 STA MEMLO+1 ; and MSB thereof
1740 ;
1750 ; and that's all we have to do!
1760 ;
F2F 60 1770 RTS
1780 ;
1790 ;
1800 ;;;;;;;;;;;;;;;;;;;;;;;;;;;
1810 ;
1820 ; This entry point is provided
1830 ; so that BASIC can reconnect
1840 ; the driver via a USR(RECONNECT)
1850 ;
1860 RECONNECT
F30 68 1870 PLA
F31 F0CD 1880 BEQ LOADANDGO ; No parameters, I hope
F33 A8 1890 TAY
1900 PULLTHEM
F34 68 1910 PLA
F35 68 1920 PLA ; get rid of a parameter
F36 88 1930 DEY
F37 D0FB 1940 BNE PULLTHEM ; and pull another
F39 F0C5 1950 BEQ LOADANDGO ; go reconnect
1960 ;

```

A sample device driver for Atari's OS The driver itself

```

1F3B      1970      .PAGE "The driver itself"
          1980 ;
          1990 ; Recall that all drivers must
          2000 ; be connected to OS through
          2010 ; a driver routines address table.
          2020 ;
          2030 MDRIVER
1F3B 4C1F   2040      .WORD MOPEN-1 ; The addresses must
1F3D 6F1F   2050      .WORD MCLOSE-1 ; ...be given in this
1F3F 921F   2060      .WORD MGETB-1 ; ...order and must
1F41 851F   2070      .WORD MPUTB-1 ; ...be one (1) less
1F43 9F1F   2080      .WORD MSTATUS-1 ; ...than the actual
1F45 491F   2090      .WORD MXIO-1 ; ...address
1F47 4C4A1F 2100      JMP  MINIT ; This is for safety only
          2110 ;
          2120 ; For many drivers, some of these
          2130 ; routines are not needed, and
          2140 ; can effectively be null routines
          2150 ;
          2160 ; A null routine should return
          2170 ; a one (1) in the Y-register
          2180 ; to indicate success.
          2190 ;
          2200 MXIO
          2210 MINIT
1F4A A001   2220      LDY  #1 ; success
1F4C 60     2230      RTS
          2240 ;
          2250 ; If a routine is omitted because
          2260 ; it is illegal (reading from a
          2270 ; printer, etc.), simply pointing
          2280 ; to an RTS is adequate, since
          2290 ; Atari OS preloads Y with a
          2300 ; 'Function Not Implemented' error
          2310 ; return code.
          2320 ;

```

A sample device driver for Atari's OS The driver function routines

```

1F4D      2330      .PAGE "The driver function routines"
          2340 ;;;;;;;;;;;;;;
          2350 ;
          2360 ; Now we begin the code for the
          2370 ; routines that do the actual
          2380 ; work.
          2390 ;
          2400 MOPEN
1F4D BD4A03 2410      LDA  ICAUX1,X ; Check type of open
1F50 2908   2420      AND  #OPUT ; Open for output?
1F52 F00D   2430      BEQ  OPENFORREAD ; No...assume for input
1F54 ADE502 2440      LDA  MEMTOP
1F57 8DD21F 2450      STA  MSTART ; We start storing
1F5A ACE602 2460      LDY  MEMTOP+1 ; ...the bytes
1F5D 88     2470      DEY ; ...one page below
1F5E 8CD31F 2480      STY  MSTART+1 ; the supposed top of mem

```

```

2490 ;
2500 ; now we join up with mode 4 open
2510 ;
2520 OPENFORREAD
1F61 ADD21F 2530 LDA MSTART ; simply move the
1F64 8DCE1F 2540 STA MCURRENT ; ...start pointer
1F67 ADD31F 2550 LDA MSTART+1 ; ...to the current
1F6A 8DCF1F 2560 STA MCURRENT+1 ; ...pointer, both bytes
2570 ;
1F6D A001 2580 LDY #STATUSOK
1F6F 60 2590 RTS ; we don't acknowledge failure
2600 ;
2610 ;
2620 ;;;;;;;;;;;;;;
2630 ;
2640 ; the routine for CLOSE of M:
2650 ;
2660 MCLOSE
1F70 BD4A03 2670 LDA ICAUX1,X ; check mode of open
1F73 2908 2680 AND #OPOUT ; was for output?
1F75 F00C 2690 BEQ MCLREAD ; no...close input 'file'
2700 ;
1F77 ADCE1F 2710 LDA MCURRENT ; we establish our
1F7A 8DD01F 2720 STA MSTOP ; ...limit so that
1F7D ADCF1F 2730 LDA MCURRENT+1 ; ...next use can't
1F80 8DD11F 2740 STA MSTOP+1 ; ...go too far
2750 ;
2760 MCLREAD
1F83 A001 2770 LDY #STATUSOK
1F85 60 2780 RTS ; and guaranteed to be ok
2790 ;
2800 ;
2810 ;;;;;;;;;;;;;;
2820 ;
2830 ; This routine puts one byte
2840 ; to the memory for later
2850 ; retrieval.
2860 ;
2870 MPUTB
1F86 48 2880 PHA ; save the byte to be PUT
1F87 20B51F 2890 JSR MOVECURRENT ; get ptr to zero page
1F8A 68 2900 PLA ; the byte again
1F8E A000 2910 LDY #0
1F8D 91E0 2920 STA (FR1),Y ; put the byte, indirectly
1F8F 20C01F 2930 JSR DECCURRENT ; point to next byte
1F92 60 2940 RTS ; that's all
2950 ;
2960 ;;;;;;;;;;;;;;
2970 ;
2980 ; routine to get a byte put
2990 ; in memory before.
3000 ;
3010 MGETB
1F93 20A01F 3020 JSR MSTATUS ; any more bytes?
1F96 B007 3030 BCS MGETRTS ; no...error
1F98 A000 3040 LDY #0
1F9A B1E0 3050 LDA (FR1),Y ; yes...get a byte
1F9C 20C01F 3060 JSR DECCURRENT ; and point to next byte
3070 MGETRTS

```



```

1F9F 60      3080      RTS
              3090 ;
              3100 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
              3110 ;
              3120 ; check the status of the driver
              3130 ;
              3140 ; this routine is only valid
              3150 ; when READING the 'file'...
              3160 ; "M:" never gets errors when
              3170 ; writing.
              3180 ;
              3190 MSTATUS
1FA0 20B51F 3200      JSR MOVECURRENT ; current ptr to zero page
1FA3 CDD01F 3210      CMP MSTOP      ; any more bytes to get?
1FA6 D009   3220      BNE MSTOK      ; yes
1FA8 CCD11F 3230      CPY MSTOP+1    ; double chk
1FAB D004   3240      BNE MSTOK      ; yes, again
1FAD A088   3250      LDY #STATUSEOF ; oops...
1FAF 38     3260      SEC              ; no more bytes
1FB0 60     3270      RTS
              3280 ;
              3290 MSTOK
1FB1 A001   3300      LDY #STATUSOK  ; all is okay
1FB3 18     3310      CLC              ; flag for MGETB
1FB4 60     3320      RTS

```

A sample device driver for Atari's OS
 Miscellaneous subroutines

```

1FB5      3330      .PAGE "Miscellaneous subroutines"
              3340 ;
              3350 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
              3360 ;
              3370 ; finally, we have a couple of
              3380 ; short and simple routines to
              3390 ; manipulate MCURRENT, the ptr
              3400 ; to the currently accessed byte
              3410 ;
              3420 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
              3430 ;
              3440 ; MOVECURRENT simply moves
              3450 ; MCURRENT to the floating
              3460 ; point register, FR1, in
              3470 ; zero page. FR1 is always
              3480 ; safe to use except in the
              3490 ; middle of an expression.
              3500 ;
              3510 MOVECURRENT
1FB5 ADCE1F 3520      LDA MCURRENT
1FB8 85E0   3530      STA FR1      ; notice that we use
1FBA ACCF1F 3540      LDY MCURRENT+1 ; both the A and
1FBD 84E1   3550      STY FR1+1    ; Y registers...this
1FBF 60     3560      RTS          ; is for MSTATUS use
              3570 ;
              3580 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
              3590 ;
              3600 ; DECCURRENT simply does a two
              3610 ; byte decrement of the MCURRENT
              3620 ; pointer and returns with the

```

```

3630 ; Y register indicating OK status.
3640 ; NOTE that the A register is
3650 ; left undisturbed.
3660 ;
3670 DECCURRENT
1FC0 ACCE1F 3680 LDY MCURRENT ; check LSB's value
1FC3 D003 3690 BNE DECLOW ; if non-zero, MSB is ok
1FC5 CECF1F 3700 DEC MCURRENT+1 ; if zero, need to bump MSB
3710 DECLOW
1FC8 CECE1F 3720 DEC MCURRENT ; now bump the LSB
1FCB A001 3730 LDY #STATUSOK ; as promised
1FCD 60 3740 RTS

```

A sample device driver for Atari's OS
RAM usage and clean up

```

1FCE 3750 .PAGE "RAM usage and clean up"
3760 ;
3770 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
3780 ;
3790 ; END OF CODE
3800 ;
3810 ;
3820 ; Now we define our storage
3830 ; locations.
3840 ;
3850 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
3860 ;
3870 ;
3880 ; MCURRENT holds the pointer to
3890 ; the next byte to be PUT or GET
1FCE 0000 3900 MCURRENT .WORD 0
3910 ;
3920 ; MSTOP is set by CLOSE to point
3930 ; to the last byte PUT, so GET
3940 ; won't try to go past the end
3950 ; of data.
1FD0 0000 3960 MSTOP .WORD 0
3970 ;
3980 ; MSTART is derived from MEMTOP
3990 ; and points to the first byte
4000 ; stored. The bytes are stored
4010 ; in descending addresses until
4020 ; MSTOP is set by CLOSE.
1FD2 0000 4030 MSTART .WORD 0
4040 ;
4050 ; DRIVERTOP becomes the new
4060 ; contents of MEMLO
2000 4070 DRIVERTOP = *+$$FF&$$FF00
4080 ; (sets to next page boundary)
4090 ;
4100 ;
4110 ; The following is how you make
4120 ; a LOAD-AND-GO file under
4130 ; Atari's DOS 2
4140 ;
1FD4 4150 *= $2E0
02E0 001F 4160 .WORD LOADANDGO
4170 ;

```

```

4180 ;
4190 .END

```

A sample device driver for Atari's OS
RAM usage and clean up

=034A ICAUX1	=0008 OPDUT	=02E7 MEMLO	=02E5 MEMTOP
=00E0 FR1	=0001 STATUSOK	=0088 STATUSEOF	=031A HADABS
=0100 HIGH	=00FF LOW	1F00 LOADANDGO	1F02 SEARCHING
1F11 EMPTYFOUND	1F25 MINSTALLED	1F3B MDRIVER	=2000 DRIVERTOP
1F30 RECONNECT	1F34 PULLTHEM	1F4D MOPEN	1F70 MCLOSE
1F93 MGETB	1F86 MPUTB	1FA0 MSTATUS	1F4A MXIO
1F4A MINIT	1F61 OPENFORREAD	1FD2 MSTART	1FCE MCERRENT
1F83 MCLREAD	1FD0 MSTOP	1FB5 MOVECURRENT	1FC0 DECCURRENT
1F9F MGETRTS	1FB1 MSTOK	1FC8 DECLOW	

Program 2.

```

100 REM listing of a program to print token values
101 REM and their ATASCII equivalents
200 ? 'The STATEMENT Token List' : ?
210 ADDR = 42161 : SKIP = 2 : TOKEN = 0
220 GOSUB 1000 : REM call the token printer
300 ? 'The OPERATOR Token List' : ?
310 ADDR = 42979 : SKIP = 0 : TOKEN = 16
320 GOSUB 1000 : REM again call to print tokens
400 END

```

```

1000 REM Subroutine to print a keyword table
1001 REM On entry:
1002 REM   ADDR = the address of the keyword table
1003 REM   SKIP = number of bytes to skip
1004 REM           between keyword strings
1005 REM   TOKEN = the starting token number for
1006 REM           this table
1007 REM
1050 IF NOT PEEK(ADDR) THEN ??:RETURN

```

[note: both tables end with a zero byte]

```

1060 PRINT TOKEN, : REM the token number
1100 REM Print the ATASCII string for this token
1110 BYTE = PEEK(ADDR) : ADDR = ADDR+1
1120 IF BYTE < 128 THEN ? CHR$(BYTE); : GOTO 1100
1130 PRINT CHR$(BYTE-128) : REM last character
      in keyword has upper bit on
1140 ADDR = ADDR + SKIP : REM an address for stmts
1150 TOKEN = TOKEN + 1 : REM to next keyword
1160 GOTO 1000

```

THE ATARI® GAZE



INSIGHT: Atari

Bill Wilkinson
Optimized Systems Software
Cupertino, CA

This month marks the end of my series on Atari I/O. That certainly doesn't mean that we won't continue to discuss assembly language I/O of related topics; it simply means that I feel I have finished my formal presentation of the material. Again, I strongly urge you to purchase the *Atari Technical User's Notes* (available from Customer Service, 1340 Bordeaux Ave., Sunnyvale, CA 94086, for \$30, including shipping). There is a lot of detail in those "notes," including much that I have glossed over. I hope that my presentation, though, has served as a usable introduction to the subject.

Also this month, I give you a method for creating relocatable assembly language programs (and a method to then load them). We use the loader to implement our "M:" driver from last month, completely via BASIC (thus making it usable for those of you not yet into assembly language...and it is usable).

Finally, we continue our discussion of how BASIC works. *De Re Atari*, and the serialized version thereof which appears in this month's *BYTE*, does a good job of discussing the *how* of BASIC's syntax; we will delve into the *why*.

Atari I/O, Part 4: GRAPHICS

Errata! Before we get started on this month's topic, I must report an error I made in **COMPUTE!** #18. On page 100, in Table 1, under the "Note" pertaining to ICBLL/ICBLH, I stated that the length is decremented by one for each byte transferred. Actually, Atari's OS is smarter than that: upon return from GET/PUT RECORD (text or binary) ICBLL/ICBLH contain a count of the number of bytes successfully transferred. This result is eminently usable (e.g., in copying records or even whole files), and perhaps we will have a program here soon that demonstrates its use.

On with the new: this whole series started as a result of a comment that I read which said something like "Atari graphics from assembly language are hard to do — you have to know about display

lists, vertical blank interrupts, etc." Knowing how BASIC does graphics for its users I said, "Nonsense! It's easy! Someone should show *how* easy!" And Richard Mansfield, of **COMPUTE!**, said, "Gee, I wonder who we could get..." Ahem.

If what you are trying to do is write an improved version of Eastern Front or Pacman or some other such pioneering project, then you need to know everything ever published and then some. *But*, if what you want is simply a way to transfer what you have learned or written using BASIC into a reasonably simple set of assembly language routines, read on.

Remember, BASIC does *all* its graphics and I/O via Atari's OS. BASIC knows nothing of graphics modes, display lists, character sets, color registers, etc. (True, BASIC A+ does its own thing with Player/Missile Graphics, but that's only because Atari's OS doesn't know about PMG.) So, anything done with standard BASIC statements can be duplicated *easily* in assembly language. To demonstrate the truth of this, Figure 1 contains a list of the seven BASIC graphics statements together with a note on how each is accomplished.

Accompanying this article is a listing of my proposal for a set of standard routines to be used by assembly language programmers when interfacing to OS graphics. These routines duplicate, as far as practicable, the statements used to do BASIC graphics. The listing clearly calls out ENTRY and EXIT parameters for each routine (i.e., register usage), so study it carefully.

As a very simple example of the routines' usage, I offer a program fragment that is written in both BASIC and assembly language:

GRAPHICS 3	LDA #3
	JSR GRAPHICS
COLOR 3	LDA #3
	JSR COLOR
PLOT 10,10	LDX #10
	LDA #0
	LDY #10
	JSR PLOT
DRAWTO 25,15	LDX #25
	LDA #0
	LDY #15
	JSR DRAWTO
SETCOLOR 2,0,14	LDX #2
	LDA #0
	LDY #14
	JSR SETCOLOR

Before leaving this topic, some notes on the

routines might be helpful: since the A-register will be zero upon entry to PLOT, DRAWTO, LOCATE, and POSITION for all graphics modes except GRAPHICS 8 (or 24), placing a LDA #0 in the beginning of POSITION would save code for anyone not using mode 8. Remember, Atari's "S:" driver can accomodate GRAPHICS 0 through 11 and 17 through 24. Adding 32 (\$20) to any graphics mode (at the time of the call to GRAPHICS) will suppress the erasure of the screen. (I haven't figured out a use for this yet, but it's nice to know it's there.)

Obviously, one could save time (and sometimes space) by performing COLOR and SETCOLOR and POSITION via simple stores (e.g., STA), but there is a certain structuring and elegance that goes with the use of the routines. The graphics routines listed herein were assembled in the \$600 page of memory, a much overworked location. I would hope that you would take the time to type them in to your assembler/editor and include them directly in future programs (EASMD users may .INCLUDE them indirectly). I really would appreciate hearing of your successes (or failures, if any) using these routines.

So far, no assembler available for the Atari produces relocatable, linkable object files (and, from what I have heard, neither will Atari's Macro Assembler). When we produced BASIC A+ and EASMD, we wanted them to move themselves to the top of memory, so we re-invented a scheme I have seen in several incarnations before: Assemble the program twice, setting the origin for any portion(s) to be relocated one page (256 bytes) higher for the second assembly, producing two object files. Write a program that compares the two objects and notes all locations that differ by one (differing by any other amount is an error). Produce a table (or bit map, or ...) of all these differences. At relocatable load time, read in the first object file (to where it is to be relocated) and use the table to change all the bytes which need to be relocated.

The system is a kludge, but a very effective one. It has a few limitations: you still don't have linkable object files, you must relocate in full page increments (i.e., multiples of 256 bytes), and you have to have some place safe to put the relocating loader. Are you willing to live with those limits? Then try this.

I present here three BASIC programs together with instructions for their use. The first program, MAKEREL (Program 1), seems to be to be perfectly adequate as is, written in BASIC. It's a little slow, but one only uses it when ready to create a new relocatable object file. The other two programs, LOADREL.A and LOADREL.B (Programs 2 and 3), could be advantageously rewritten in assembly

language. They are presented here in BASIC because (1) this method fulfills the requirement for a "safe place" for the loader and (2) by presenting them in BASIC they can be used by those not yet ready to tackle assembly language and (3) it was easier for me.

The instructions below presume the use of the Atari Assembler/Editor or the OSS EASMD, but they can be easily adapted to most systems that produce Atari DOS-compatible object files.

How To Use The Relocator Programs

- 1) Write, assemble, and debug your code using some fixed address(es).
- 2) Ensure that your code is all in one piece (i.e., there is only one `*=`, at the beginning of the code segment).
- 3) Origin your code on an even page boundary (i.e., use `*=$hh00`, where 'hh' specifies any page from 02 through FE). Assemble the code into an object file on disk named "OBJECT1" (use `ASM ,#D:OBJECT1`).
- 4) Change your origin to one page higher in memory (`*=$nn00`, where 'nn' = 'hh' + 1). Assemble the code to "OBJECT2" (`ASM ,#D:OBJECT2`).
- 5) Run the MAKEREL program. It will produce the file "DATA.REL".
- 6) Adjust the value of the variable NUMBEROF-PAGES in both LOADREL.A and LOADREL.B (Programs 2 and 3) to reflect the number of 256-byte pages needed by your routine. **SAVE** the adjusted versions.
- 7) Anytime you want to load your routine, simply use `RUN "D:LOADREL.A"`.

Notes

- A. Generally, it's a good idea to have your routine start execution at the origin (`*=`) point. Then you can invoke it from BASIC via `USR(PEEK(128) + 256 *(PEEK(129) - NUMBEROFPAGES))`
- B. If you `RUN "D:LOADREL.A"` again without hitting RESET, it will load another copy above the first. Not too neat, *but* the advantages of being able to thus load several different modules should be obvious!
- C. LOADREL.B performs an `ENTER "D: DATA.REL"`. Rather than waiting for the `ENTER` each time, you may **SAVE** the resultant program (after taking out the `ENTER` line) for a slightly faster load of a specific module.

Finally, we offer Program 4 which may be added to LOADREL.B to produce a relocatable load of last month's "M:" driver. (Again, be sure to delete the `ENTER` line from LOADREL.B.)

For once, I haven't forgotten you cassette

users. If you enter LOADREL.A (carefully, please!) and **CSAVE** it (or **SAVE"C:"**) on a blank tape you need only change the last line to read **RUN "C:"**. Then **NEW** and enter LOADREL.B, leaving out the **ENTER** line, but including the listing of Program 4. Use **SAVE"C:"** (do NOT use **CSAVE...** it won't work!) to place the resultant combination on the tape after LOADREL.A (and, of course, you could then follow on the same tape with a program of your own). You may now enjoy the "M:" driver via this tape by **CLOADing** and **RUNning** the first program (or use **RUN"C:"** if you used **SAVE"C:"**, my own preference for all but the largest programs).

MAKEREL could also be adapted to cassette usage, though not without difficulty and/or a relatively large amount of memory. Obviously, these programs can be improved upon tremendously by simply adding, for example, flexibility of file name. But my intention was to present something as simple and straightforward as possible, in the hopes that everyone would find it readable and useful. Obviously, my techniques could be adapted to other machines (does the PET have a relocating assembler?), so adapt away (and be sure to send **COMPUTE!** the results to share with the rest of us). On to lighter subjects.

Inside Basic, Part 2: The Why Of Syntaxing

Last month I presented a program to print out the keywords of BASIC. If you took the time to enter and run that program, you saw some strange things in the printout of the operators. But there was a method to our madness, as you will see.

Let us examine the tokenized (internal) form of the following line:

```
1025 PRINT "HI THERE", THIS * (3 + IS(FUN))
: STOP
```

Assuming that we had just previously **NEWed**, the tokenized form of that line is as follows (all numbers in decimal):

```
01 04 36 33 32 15 08 72 73 32 84 72 69 18 128
36 43 14 64 03 00 00 00 00 37 129 56 130 44
44 20 36 38 22
```

Now that isn't too terribly useful or readable, so let's examine the tokens one at a time:

```
01 04      This is the line number (4*256 + 1 = 1025)
           in standard 6502 form.
36         This is the line length, including the line
```

number and this byte.

33	Statement length of the first statement. Actually, this is the displacement to the beginning of the next statement (from the beginning of the line).
32	The token for PRINT. Check the output of the keyword printing program from last month.
15	A special token that says a string constant follows.
08 72 73 32 74 72 69 82 69	The string constant consists of a byte that gives the length of the string followed by the characters of the string. Note that the quotes have disappeared.
18	The comma, tokenized.
128	Our first variable! Operator tokens over 127 are variables. The variable number (in the variable table) is 128 less than the token value. This variable is THIS .
36	The multiplication operator.
43	One variety of left parenthesis. This one is a normal or expression left parenthesis.
14	Another special token (actually, number 2

Figure 1.

BASIC Statement	Action performed
GRAPHICS g	If bit 4 (\$10) of 'g' is on, this is the same as OPEN #6, 12, g-16, "S:" If the bit is off, this is the same as OPEN #6, 16 + 12, g, "S:" (Note: the fifth bit, \$20, of 'g' should be copied into AUX1, the OPEN mode.) Simply saves 'c' in a safe place. Places 'h' in locations \$55 and \$56 (LSB,MSB)
COLOR c	Places 'v' in location \$54
POSITION h,v	Performs a POSITION h,v and then Performs a PUT #6,c (where 'c' is the color saved by COLOR)
PLOT h,v	Performs a POSITION h,v and then Performs a GET #6,c
LOCATE h,v,c	Performs a POSITION h,v and then Does a POKE 763, c ('c' is the COLOR saved, as above) and then
DRAWTO h,v	Performs an XIO 17, #6, 12, 0, "S:"
SETCOLOR r,h,lu	Is equivalent to POKE 708 + r, h*h16 + lu

Note: FILL may be performed from assembly language by following exactly the same sequence specified in the *Basic Reference Manual*, using XIO 18, etc.

Program 1: MAKEREL

```
100 REM *** OPEN ALL 3 FILES ***
110 OPEN #1,4,0,"D:OBJECT1"
120 OPEN #2,4,0,"D:OBJECT2"
130 OPEN #3,8,0,"D:DATA.REL"
150 REM *** INITIALIZE VARIABLES ***
160 LINE=10000
```

of 2), says a numeric constant follows.

64 03 00 00 The constant, in Atari
00 00 BASIC internal floating
point form. This is
unique, as we shall see
soon.

37 An addition operator.

129 The variable **IS** (already
known to be an array,
though it has not yet
been DIMensioned).

56 Another left paren-
thesis. This one is called
an "array left paren" in
the BASIC source listing.
We will later see why it
is distinct.

130 Our last variable, **FUN**.

44 44 Two right parentheses.
Strange, they are both
the same.

20 Our End-Of-Statement
token, otherwise known
as a colon.

36 The statement end dis-
placement for the second
statement on this line.

38 The token for **STOP**.
Again, refer to the key-
word listing program.

22 An End-Of-Line token,
otherwise known as a
RETURN.

Wasn't that fun? For a maso-
chist? Hopefully, you are asking
questions that begin with "Why."

Why tokenize at all? For com-
pactness: in our example we saved
six bytes over a straight source
line. For speed: it is much faster (at
run-time) to discover that, for ex-
ample, 32 means "PRINT" than it
would be if we had to examine the
letters "P", "R", "I", "N", "T" for a
keyword match. Because token-
izing is almost an automatic by-
product of syntaxing.

Why syntax-check at entry?
Because it is embarrassing to give a
program to someone, have them
run it, and get a SYNTAX ERROR
message at line 23776 (the line that
handles disk full conditions, which
we never got to when we were
testing). Because it makes pro-
gram entry so much easier for be-



ALI BABA and the Forty Thieves **QUALITY SOFTWARE**

Encounter sultans, thieves, fierce and friendly creatures as you guide your alter ego, Ali Baba, through the thief's mountain den in an attempt to rescue the beautiful Princess. Treasure, magic, and danger await you! Up to seventeen friendly characters can join together to rescue the Princess-all in high resolution color graphics. Every adventure is different and exciting. Ali Baba also features a **SAVE GAME** option for resumed play later. Keyboard or joystick.
Cat No. 3391 Atari, 32K, disk, machine language
\$32.95

VERBATIM DATLIFE DISKETTES

Perfect for ATARI, APPLE, TRS-80, or any 5¼" soft-sector computer (35 or 40 track). Includes a built-in hub protector ring.
Cat No. 1147
\$28.00

GHOST HUNTER ARCADE PLUS

Your mission in **GHOST HUNTER** is simple-rid the mansion on Huckleberry Hill of ghosts...before they get you! Features high resolution, fast-paced action as you race around in one of up to 16 different mazes. **GHOST HUNTER** is a one or two player game-play solo or head-to-head against another player. For Atari 800 OR 400!

Cat. No. 3341 Atari 16K, cass., joystick, machine language **\$29.95**
Cat. No. 3342 Atari 16K, disk, joystick, machine language **\$34.95**

HOW TO ORDER

Mention this ad and **WE PAY SHIPPING** (UPS ground-USA)! Call or write. Pay by check, M/C, Visa, or COD (add \$1.40 for COD). Offer expires March 1, 1982.

**!!!! FREE SHIPPING FREE SHIPPING
FREE SHIPPING !!!!!**

HW ELECTRONICS

19511 Business Center Dr. Dept. G2
Northridge, CA 91324
(800) 423-5387 (213) 886-9200


```

170 DCNT=0
200 REM *** STRIP HEADER ($FFFF) WORD ***
220 GET #1,FF:GET #1,FF
230 REM STRIP HEADER AND ADDRESSES FROM FILE2
240 GET #2,FF:GET #2,FF:REM HEADER
250 GET #2,FF:GET #2,FF:REM START ADDRESS
260 GET #2,FF:GET #2,FF:REM END ADDRESS
300 REM *** PROCESS ADDRESSES ***
310 GET #1,LOW:GET #1,FIRSTHIGH:FIRST=LOW+256*FIRSTHIGH
320 GET #1,LOW:GET #1,HIGH:LAST=LOW+256*HIGH
400 REM *** READY TO PRODUCE OUTPUT ***
410 FOR ADDR=FIRST TO LAST
420   IF DCNT=0 THEN PRINT #3;LINE;" DATA ";;LINE=LINE+10
430   GET #1,B1:GET #2,B2
440   IF B1=B2 THEN 480
450   IF B2<>B1+1 THEN PRINT "BAD RELOCATION":STOP
460   B1=B1-FIRSTHIGH:REM THE RELOCATION FACTOR
470   PRINT #3:"*";:REM AND FLAG THIS BYTE
480   PRINT #3;B1;
490   DCNT=DCNT+1
500   IF DCNT<=9 THEN PRINT #3;" ";
510   IF DCNT>9 THEN DCNT=0:PRINT #3
520   NEXT ADDR
530 REM *** CLEAN UP ***
540 IF DCNT=0 THEN PRINT #3;LINE;" DATA ";
550 PRINT #3;"="
560 PRINT #3;"GOTO 500"
580 CLOSE #1:CLOSE #2:CLOSE #3
590 END

```

Program 2: LOADRELA

```

10 REM *** THIS IS LOADREL.A ***
20 REM (THIS SIMPLY SETS UP MEMORY FOR LOADREL.B)
30 NUMBEROFFPAGES=1:REM CHANGE THIS AS NEEDED
40 SIZE=256*NUMBEROFFPAGES
100 REM *** SEE COMPUTE! #19 ***
110 LET LOMEM=743:MEMLOW=128
120 LADDR=PEEK(LOMEM):HADDR=PEEK(LOMEM+1)
129 REM -- LINE 130 ENSURES THAT 1K BYTES STARTS ON PAGE BOUNDARY --
130 IF LADDR<>0 THEN LADDR=0:HADDR=HADDR+1
140 ADDR=LADDR+256*HADDR
150 ADDR=ADDR+SIZE
160 HADDR=INT(ADDR/256):LADDR=ADDR-256*HADDR
170 POKE LOMEM,LADDR:POKE LOMEM+1,HADDR
180 POKE MEMLOW,LADDR:POKE MEMLOW+1,HADDR:RUN "D:LOADREL.B"

```

Program 3: LOADREL.B

```

100 REM *** THIS IS LOADREL.B ***
110 REM
120 REM THIS PROGRAM DOES THE ACTUAL RELOCATABLE LOAD
130 REM
140 DIM TEMP$(10)
150 NUMBEROFFPAGES=1:REM ADJUST TO SAME AS LOADREL.A
200 REM AGAIN, SEE COMPUTE! #19
210 LET LOMEM=743:MEMLOW=128
220 POKE LOMEM,PEEK(MEMLOW):POKE LOMEM+1,PEEK(MEMLOW+1)
300 REM RPAGE IS THE MEMORY PAGE WHERE WE RELOCATE TO
310 RPAGE=PEEK(MEMLOW+1)-NUMBEROFFPAGES
330 REM OBVIOUSLY, THIS VALUE SHOULD MATCH THE MEMORY
340 REM RESERVED IN 'LOADREL1.SAV'
350 ADDR=RPAGE*256:REM STARTING ADDR OF LOAD

```


ginners, particularly kids. Because like it.

Why one-byte variable numbers? Again, for speed and compactness. Use variable names as long as you like: only the first usage eats up any more memory than a single-character, undecipherable variable name. There are disadvantages: a maximum of 128 different variables, a misspelled variable name can't be purged from the variable table without LISTing and reENTERing. On the whole, a very wise choice (*I can say that, it's one part of Atari BASIC I didn't design into the specs*).

Why internalized numeric constants? For speed. Period. Well, maybe for simplicity at run-time, but that's only a maybe. Did you know that numeric constants in Atari BASIC actually execute faster than variables? Write a timing loop and prove it to yourself.

Why line length bytes? Do you need them if you have statement length bytes? We don't need them, but they make line skipping (as when we are executing a GOTO) faster than it would be if we had to skip individual statements.

Why statement length bytes? Given that you have line length bytes? This one is harder to answer, because it has to do with how we execute GOSUB/RETURN, etc. I will leave that for a later article, but I will note that these bytes were extremely helpful when it came to implementing the IF...ELSE...ENDIF structure in BASIC A+.

Why decimal floating point? Because it is easier for beginners to understand (try PRINT 123.123-123 using Applesoft) and is obviously preferable for money applications. Actually, our decimal add and subtract are faster than the corresponding binary routines. Admittedly, multiply suffers a little and divide suffers a lot.

Why different kinds of left parentheses? Why several kinds of equal sign? Because it's easy for the syntaxer to see the different



GALACTIC CHASE™

The aliens have swept undefeated across the galaxy. You are an enterprising star ship captain—the final defender of space.

As the aliens attack, you launch a deadly barrage of missiles. Flankers swoop down on your position. Maneuvering to avoid the counterattack, you disintegrate their ships with your magnetic repellers.

As your skill improves, the attackers increase their speed. And as a last resort, the aliens use their invisible ray to slow the speed of your missile launcher.

GALACTIC CHASE provides Atari owners with the most challenging one or two person game in the galaxy.



Atari 400/800 16k. Written in machine language. Requires joysticks.

Payment: Personal Checks—allow three weeks to clear.

American Express, Visa, & Master Charge—include all numbers on card. Please include phone number with all orders. 24.95 for cassette or 29.95 for disk plus 2.00 shipping. Michigan residents add 4%.

Check the dealer in your local galaxy. Dealer inquiries encouraged.

Galactic Chase © 1981 Stedek Software.

SPECTRUM
COMPUTERS

Dept C.
26618 Southfield
Lathrup Village, MI. 48076
(313) 559-5252

```

400 REM *****
410 REM *   GET THE RELOCATION DATA *
420 REM *****
450 ENTER "D:DATA.REL"
500 REM *** THE ENTER BRINGS US HERE ***
510 READ TEMP$
520 IF TEMP$(1,1)="=" THEN END
530 IF TEMP$(1,1)<>"*" THEN POKE ADDR,VAL(TEMP$):GOTO 550
540 POKE ADDR,VAL(TEMP$(2))+RPAGE:REM RELOCATION
550 ADDR=ADDR+1:GOTO 510

```

Program 4: DATA.REL

```

520 IF TEMP$(1,1)="=" THEN 1000
1000 REM LINE 1010 IS USED TO INITIALIZE THE M: DRIVER
1010 JUNK=USR(RPAGE*256+48)
1020 END
10000 DATA 162,0,189,26,3,240,10,201,77,240
10010 DATA 26,232,232,232,208,242,96,169,77,157
10020 DATA 26,3,169,59,157,27,3,169,*0,157
10030 DATA 28,3,169,0,157,29,3,169,0,141
10040 DATA 231,2,169,*1,141,232,2,96,104,240
10050 DATA 205,168,104,104,136,208,251,240,197,76
10060 DATA *0,111,*0,146,*0,133,*0,159,*0,73
10070 DATA *0,76,74,*0,160,1,96,189,74,3
10080 DATA 41,8,240,13,173,229,2,141,210,*0
10090 DATA 172,230,2,136,140,211,*0,173,210,*0
10100 DATA 141,206,*0,173,211,*0,141,207,*0,160
10110 DATA 1,96,189,74,3,41,8,240,12,173
10120 DATA 206,*0,141,208,*0,173,207,*0,141,209
10130 DATA *0,160,1,96,72,32,181,*0,104,160
10140 DATA 0,145,224,32,192,*0,96,32,160,*0
10150 DATA 176,7,160,0,177,224,32,192,*0,96
10160 DATA 32,181,*0,205,208,*0,208,9,204,209
10170 DATA *0,208,4,160,136,56,96,160,1,24
10180 DATA 96,173,206,*0,133,224,172,207,*0,132
10190 DATA 225,96,172,206,*0,208,3,206,207,*0
10200 DATA 206,206,*0,160,1,96,0,0,0,0
10210 DATA 0,0,=

```

Program 5: Graphics Routines, Equates

```

0000      1010      .PAGE "EquateS. etc."
          1020 ;
          1030 ; CIO EQUATES
          1040 ;
E456      1050 CIO      =      $E456      ; Call OS thru here
0342      1060 ICCOM    =      $342      ; COMmand to CIO in IoCb
0344      1070 ICBADR   =      $344      ; Buffer or filename ADdRess
0348      1080 ICBLN    =      $348      ; Buffer LENgth
034A      1090 ICAUX1   =      $34A      ; AUXilliary byte # 1
034B      1100 ICAUX2   =      $34B      ; AUXilliary byte # 2
          1110 ;
0003      1120 COPN     =      3          ; Command OPen
000C      1130 CCLOSE   =      12         ; Command CLOSE
0007      1140 CGBINR   =      7          ; Command Get BINary Record
000B      1150 CPBINR   =      11         ; Command Put BINary Record
0011      1160 CDRAW    =      17         ; Command DRAWto
0012      1170 CFILL    =      18         ; Command FILL (not used in this demo)
          1180 ;
0004      1190 OPIN     =      4          ; OPen for INput
000B      1200 OPOUT    =      8          ; OPen for OUTput
          1210 ;
          1220 ;
          1230 ; EQUATES used by the S: driver and

```

kinds of equal signs in, for example, $LET A = B = C + D\$ = E\$$.

Sure, we could tell the difference at run time from context, but why should we when it's so easy to distinguish between a 45 and a 34 and a 52?

Why doesn't Atari BASIC have string arrays? I really didn't want to put this question in, but I wanted to save myself the letters and threatening phone calls. The best reason is that it was a choice of string arrays or syntax checking. (Obviously, I like the choice.) Other rationales include the fact that Atari was aiming for the educational market, where the HP2000 (with 72-character, Atari-style strings) was the *de facto* standard.

My personal favorite reasons are twofold: (1) anything you can do with string arrays you can also do with long strings (admittedly, sometimes with a little more difficulty) though the reverse is definitely not true; and (2) string arrays are unique to DEC/ Micro-soft/??? BASIC and do not appear in that form in any other of the more popular languages (e.g., FORTRAN, COBOL, PASCAL, C, FORTH, etc.). Techniques learned with long strings are portable to these other languages: techniques involving string arrays are, at best, difficult to transfer. Finally, long strings as implemented on the Atari have some unique advantages not immediately obvious. I hope to explore some of these advantages in future columns.

TOLL FREE
Subscription
Order Line
800-345-8112
In PA 800-662-2444

SWIFTY SOFTWARE TOP RATED PRODUCTS FOR ATARI



NEW! SAVE \$5 HARDWARE DISK SENTRY™

An intelligent digital accessory for your ATARI 810 Disk Drive, lets you selectively write data to both sides of single sided and write protected disks. DISK SENTRY cannot harm your drive or disks. Installs and removes easily; no soldering required. DISK SENTRY's LED signals system status, preventing accidental erasure of data. DISK SENTRY is a convenient push button write-protect override which can pay for itself with your first box of disks. \$39.95 + \$2.50 Shipping and Handling.

ARCADE GAMES

24K Disk; 16K Cassette; Joystick required
Add these popular HIGH RESOLUTION, REAL-TIME, ANIMATED games to your software arsenal. Get FAST ACTION and FULL SOUND GRAPHICS that take advantage of the unique features of your ATARI. Enjoy challenge that requires strategy and skill.

SPACE CHASE™

Fly against intelligent invader clones. Arm yourself with Nuclear Defense Charges and play with or without Defense Shields. Enjoy this action-packed multicolor space odyssey. Displays top score, number of planets saved and number of galaxies conquered. \$14.95 cassette; \$19.95 disk

TIMEBOMB™

Meet the challenge of this fast moving animated race against time, enemy aircraft and enemy bombs as you attempt to disarm timebombs set to explode ammunition depots. Avoid aircraft of varying sizes and speeds — and their bombs. Choose one of ten Day or Night Missions. Use from one to four Joysticks. Any number can play; top players listed on scoreboard. \$14.95 cassette; \$19.95 disk

NEW! AND MORE GAMES..... TRIVIA TREK™

Unlimited fun and lots of laughs for one or two players. Five hundred questions and two thousand multiple choice answers are supplied on the master diskette. A powerful datafile handling program allows creation of your own trivia questions and answers. Features include: Player Missile Graphics, user or random selection of subjects and numerous comical answer choices. This DISK ONLY package comes complete with user instructions. An incredible value for only \$29.95. Requires 32K and disk drive.

NEW! FUN "n" GAMES #1™

WORDGAMES, POSSIBLE and LEAPFROG giving you hours of fun, challenge and entertainment. WORDGAMES, two games in one, contains GUESSIT - a deductive alphabetic reasoning game for one or two players and WORDJUMBLE - a multiple word descrambling puzzle with play-on-word hints and mystery answers. Instructions show how you can substitute your own words. Use POSSIBLE to help descramble word jumble puzzles or to create your own. All letter/number combinations or permutations of input are printed to screen or optional printer. LEAPFROG is a Chinese-Checker type jumping game in which you try to position two sets of animated jumping frogs in a minimum number of moves. 16K Cassette \$19.95; 24K Disk \$24.95. Disk version of GUESSIT works with VOTRAX Type "n" TALK. A real crowd pleaser.

COMING SOON! Space Shuttle Adventure Series™

Real-time Space Flight Simulations

★ PERSONAL DATA MANAGEMENT FILE-IT 2™

Contains all the programs in FILE-IT plus five additional file handling and financial programs. Financial entry and report generator programs create a powerful personal accounting system while two additional utility programs provide random access updating and user controlled record selection. Subfiles may be created, merged and sorted by any field. A monthly Bar Graph program generates a visual picture of financial data on the screen and/or printer. Supports up to four disk drives as well as the AXLOW RAMDISK. Minimum requirements are 24K, 1 disk drive and an 80 column printer. Extensive documentation, supplied in a ring binder, provides clear instruction along with a tutorial on computer filing. \$49.95 + \$3.25 Shipping and Handling. AXLOW RAMDISK not required.

FILE-IT™

Use this start up database system to file and manage personal information and data. Create, sort, store and manipulate information such as appointment calendars, address, or telephone data, credit or charge records, stock investments, medical or prescription information, hobby, coupon or other types of collection information....and more. With printer you get 1 or 2 across mailing labels, disk jacket inventory covers and neatly written copy of all your data files. Comes with well documented instruction manual explaining basics of computer filing. Fast and easy to use. Holds over 300 records in 40K. Requires minimum of 24K and 1 disk drive. Printer optional. \$34.95 (Disk Only)

COMING SOON! The Family Financier™

AN easy to use financial package.

UTILITIES

DISKETTE INVENTORY SYSTEM™

Use this system to gain control of your expanding disk/program inventory. Quickly get locations of single or multiple copies of your programs and all your valuable files. An invaluable tool, this system is easy and convenient to use and to update. 24K disk system required. \$24.95 Printer suggested.

SWIFTY UTILITIES

A valuable collection of programming utilities for the ATARI programmer. This DISK ONLY package includes all of Programming Aids I and additional programs designed to make programming time more efficient. Special MENU program runs both saved and listed programs. REM REMOVER eliminates REM statements so programs take less core and run faster. PRINT 825 and PRINTEPS custom print programs prepare condensed, indented and paginated program listings on your ATARI 825 or EPSON MX-80 printer. Listings identify machine code, graphics and inverse video characters. VARIABLE LIST and VARIABLE PRINT programs help you prepare alphabetized annotated list of your program variables. A delete lines utility provides convenience of line deletion while a DOS CALLER gives you convenient access to many DOS utilities while your program is in core. Disklist prepares disk jacket labels. Many of these programs work coreident with each other and with your program. Disk Drive and minimum of 24K required. \$29.95

PROGRAMMING AIDS PACKAGE I™

Four utility programs to help increase programming efficiency and learn more about your computer. RENUMBER handles references and even variables. Generates Diagnostic Tables for programming error detection. PROGRAM DECODER, DECIMAL to BCD and BCD to DECIMAL programs give you a practical way of studying internal program representation and ATARI number-conversion procedures. Comes with comprehensive user's manual. 16K cassette \$14.95; 24K disk \$19.95

SWIFTY DATALINK™

High Quality Smart Terminal Communications program. Easy to use Multi-Option, Menu Driven. Full performance uploading/downloading. Works in Duplex or Simplex modes supporting ASCII and ATASCII transmission. Printer Dump, Screen Dump and Disk Search options. Use as remote terminal. Send/receive and store programs and data files. Saves connect time charges with commercial services. Requires 24K RAM, 810 Disk Drive, 850 Interface or equivalent, 830 or other 300 Baud modem. (Printer optional) \$39.95

NEW! SWIFTY TACH MASTER™

An accurate disk speed diagnostic utility program designed specifically for ATARI 810 Disk Drives. Provides easy-to-read visual indication of the speed of any drive connected to your system. Using the accuracy of machine language, DISK DOCTOR displays five RPM readings per second with a working tachometer accurate to 1/4 RPM. Allows you to adjust your drive(s) to factory specs easily and at any time in the convenience of your own home. Comes complete with easy to follow user's manual. \$29.95

NEW! ACCESSORIES VINYL DUST COVERS

New, glove soft, vinyl dust covers for the ATARI 800 Computer, the 400 Computer and the 825 Printer. Custom made from heavy duty upholstery grade vinyl, these covers completely cover the top and sides of your valuable equipment. Do not confuse them with cheap, flimsy plastic covers available elsewhere. Accessory ports and other input/output plugs are exposed for convenience of use. Available in either black or "ATARI" beige. ATARI 400: \$9.95; ATARI 800: \$10.95; ATARI 825: \$10.95. Specify model and color. Any two covers for \$18.95. Please include \$2.50 for Shipping and Handling.

send check or money order to:

SWIFTY SOFTWARE, INC.

64 BROAD HOLLOW ROAD
MELVILLE, N.Y. 11747
(516) 549-9141

N.Y. Residents add 7 1/4 % sales tax

send for free catalogue dealer orders and c.o.d.'s accepted

©1981, 1982 Swift Software, Inc.

NOTE: ATARI® is a registered trademark of Atari Inc., a Warner Communications Company and all references to ATARI® should be so noted.

```

1240 :   the VBLANK routines
1250 :
0055 1260 HORIZONTAL = $55
0054 1270 VERTICAL = $54
02FB 1280 DRAWCOLOR = $2FB
02C4 1290 COLOR0 = $2C4
      1300 ;
      1310 ; miscellany
      1320 ;
00FF 1330 LOW = $FF
0100 1340 HIGH = $100
      1350 ;

```

Graphics routines for COMPUTE! #21
The actual routines

```

0000 1360 .PAGE "The actual routines"
      1370 ;
      1380 ; First, set the location and some miscellaneous
      1390 ; RAM usage
      1400 ;
0000 1410 *= $660
      1420 ;
0660 00 1430 SAVECOLOR .BYTE 0 ; where COLOR is saved
      1440 ;
0661 53 1450 SNAME .BYTE "S:",0 ; the filename for open
0662 3A
0663 00
      1460 ;
      1470 ;
      1480 ; GRAPHICS g
      1490 ;
      1500 ; ENTRY: A-reg contains graphics mode 'g'
      1510 ; EXIT: Y-reg has completion status
      1520 ;
      1530 GRAPHICS
0664 48 1540 PHA ; save 'g'
0665 A260 1550 LDX #6*$10 ; file 6
0667 A90C 1560 LDA #CCLOSE
0669 9D4203 1570 STA ICCOM.X
066C 2056E4 1580 JSR CIO ; First, we must close file #6
      1590 ; (we ignore any errors from the close)
      1600 ;
066F A260 1610 LDX #6*$10 ; again, file 6
0671 A903 1620 LDA #COPN ; we will open this 'file'
0673 9D4203 1630 STA ICCOM.X
0676 A961 1640 LDA #SNAME&LOW
0678 9D4403 1650 STA ICBADR.X ; we use the file name "S:"
067B A906 1660 LDA #SNAME/HIGH
067D 9D4503 1670 STA ICBADR+1.X ; by pointing to it
      1680 ;
      1690 ; all is set up for OPEN, now
      1700 ; we tell CIO (and S:) what kind of open
      1710 ;
0680 68 1720 PLA ; our saved 'g' graphics mode
0681 9D4B03 1730 STA ICAUX2.X ; is given to 'S'
      1740 ; (note that S: ignores the upper bits of AUX2)
0684 29F0 1750 AND #$F0 ; now we get just the upper bits
0686 4910 1760 EOR #$10 ; and flip bit 4
      1770 ; (Read the text. S: expects this bit inverted
      1780 ; from what normal BASIC usage is.)
0688 090C 1790 ORA #$0C ; allow read and write access (for CIO)
068A 9D4A03 1800 STA ICAUX1.X ; make CIO and S: happy
068D 2056E4 1810 JSR CIO ; and do the OPEN of S:

```



```

0690 60      1820      RTS
              1830 ;
              1840 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
              1850 ;
              1860 ; COLOR c
              1870 ;
              1880 ; ENTER: Color 'c' in A-register
              1890 ; EXIT: Unchanged
              1900 ;
              1910 COLOR
0691 8D6006 1920      STA SAVECOLOR
0694 60      1930      RTS ; exciting, wasn't it?
              1940 ;
              1950 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
              1960 ;
              1970 ; POSITION h,v
              1980 ;
              1990 ; ENTER: h (horizontal) position in X,A
              2000 ; registers (LSB,MSB)
              2010 ; v (vertical) position in Y-register
              2020 ;
              2030 ; EXIT: unchanged
              2040 ;
              2050 POSITION
0695 8655    2060      STX HORIZONTAL
0697 8556    2070      STA HORIZONTAL+1 ; read the text
0699 8454    2080      STY VERTICAL ; too simple, right?
069B 60      2090      RTS
              2100 ;
              2110 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
              2120 ;
              2130 ; PLOT h,v
              2140 ;
              2150 ; ENTER: must have done a previous COLOR call
              2160 ; X.A, and Y registers set as in POSITION
              2170 ;
              2180 ; EXIT: Y-register has completion status
              2190 ;
              2200 PLOT
069C 209506 2210      JSR POSITION
069F A260    2220      LDX #6*10 ; file 6, again
06A1 A90B    2230      LDA #CPBINR ; Command Put BINary Record
06A3 9D4203 2240      STA ICCOM,X
06A6 A900    2250      LDA #0
06A8 9D4803 2260      STA ICBLN,X
06AB 9D4903 2270      STA ICBLN+1,X ; if buffer length is zero...
06AE AD6006 2280      LDA SAVECOLOR ; then CPBINR puts one char from A-reg
06B1 2056E4 2290      JSR CIO ; and this is how we PLOT
06B4 60      2300      RTS
              2310 ;
              2320 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
              2330 ;
              2340 ; LOCATE h,v,c
              2350 ;
              2360 ; ENTER: X.A, and Y registers set up as in POSITION
              2370 ; EXIT: A-register has the LOCATED color
              2380 ; Y-register has the completion code
              2390 ;
              2400 LOCATE
06B5 209506 2410      JSR POSITION
06B8 A260    2420      LDX #6*10 ; file 6
06BA A907    2430      LDA #CGBINR ; Command Get BINary Record
06BC 9D4203 2440      STA ICCOM,X

```

```

06BF A900 2450 LDA #0
06C0 9D4803 2460 STA ICBLN,X
06C4 9D4903 2470 STA ICBLN+1,X ; if Buffer LENgth is zero,
06C7 2056E4 2480 JSR CIO ; then the character is returned in A
06CA 60 2490 RTS
2500 ;
2510 ;::::::::::::::::::::::::::::::::::::::::::
2520 ;
2530 ; DRAWTO h,v
2540 ;
2550 ; ENTER: must have done a previous PLOT
2560 ; X.A.and Y registers as in POSITION
2570 ;
2580 ; EXIT: Y-register has completion code
2590 ;
2600 DRAWTO
06CB 209506 2610 JSR POSITION
06CE AD6006 2620 LDA SAVECOLOR
06D1 8DFB02 2630 STA DRAWCOLOR ; where DRAWTO expects its color
06D4 A260 2640 LDX #6*#10 ; file 6...once more
06D6 A911 2650 LDA #CDRAW ; just a command to "S:"
06D8 9D4203 2660 STA ICCOM,X
06DB A90C 2670 LDA #0C
06DD 9D4A03 2680 STA ICAUX1,X ; insurance
06E0 A900 2690 LDA #0
06E2 9D4B03 2700 STA ICAUX2,X ; ...guaranteed to work
06E5 2056E4 2710 JSR CIO ; do the actual DRAWTO
06E8 60 2720 RTS
2730 ;
2740 ;::::::::::::::::::::::::::::::::::::::::::
2750 ;
2760 ; SETCOLOR r,hue,lum
2770 ;
2780 ; ENTER: X-register has color register 'r'
2790 ; A-register has hue
2800 ; Y-register has luminance
2810 ; EXIT: (undefined)
2820 ;
2830 SETCOLOR
06E9 0A 2840 ASL A
06EA 0A 2850 ASL A
06EB 0A 2860 ASL A
06EC 0A 2870 ASL A ; we need hue * 16
06ED 9DC402 2880 STA COLOR0,X ; save it here for a nonce
06F0 98 2890 TYA
06F1 290E 2900 AND #0E ; only luminance bits that matter
06F3 18 2910 CLC
06F4 7DC402 2920 ADC COLOR0,X ; end of the nonce
06F7 9DC402 2930 STA COLOR0,X ; and VBLANK will move this to hardware
06FA 60 2940 RTS
2950 ;
06FB 2960 .END

```

Graphics routines for COMPUTE! #21
The actual routines

=E456 CIO	=0342 ICCOM	=0344 ICBADR	=0348 ICBLN
=034A ICAUX1	=034B ICAUX2	=0003 COPN	=000C CCLOSE
=0007 CGBINR	=000B CPBINR	=0011 CDRAW	=0012 CFILL
=0004 OPIN	=0008 OPOUT	=0055 HORIZONTAL	=0054 VERTICAL
=02FB DRAWCOLOR	=02C4 COLOR0	=00FF LOW	=0100 HIGH
0660 SAVECOLOR	0661 SNAME	0664 GRAPHICS	0691 COLOR
0695 POSITION	069C PLOT	06B5 LOCATE	06CB DRAWTO
06E9 SETCOLOR			

INSIGHT: ATARI

Bill Wilkinson
Optimized Systems Software

Good news! I have finally found out how and where you will be able to obtain copies of *De Re Atari* ... and it won't even cost you your left thumb. The Atari Program Exchange now has it available for \$19.95 plus shipping. The part number for it is APX-90008, and you can order it through 800-538-1862 (800-672-1850 in California). There are several changes and improvements from earlier versions, including a section on the GTIA. One disappointment is that an appendix on random access files has been deleted. Oh well, leaves room for me to do a future article.

The How and Why articles on Atari BASIC that appeared in the last two issues were the result of requests for ways of "hooking into" BASIC, in order to add commands, etc. I am trying to gently break the news that you *can't* add commands to a RUNning program (though direct, keyboard commands can be done by intercepting keyboard input, as I presume the Eastern House "Monkey Wrench" does.). But I have been trying to lead up to *why* you can't add commands, so that people won't waste time on false leads in trying to prove me wrong.

However, I am suspending the How and Why series this month in order to take a look at the USR function. It is my belief that the USR function will give most of you access to all the added commands you could write, which lessens somewhat the impact of not being able to integrate your own commands. In addition to some suggestions on usage, this month we implement a really powerful USR function: one which will play a song (or most any kind of sound) in the background while your BASIC program continues to chug away (zapping Klingons, etc.). Naturally, there will also be the usual mix of tricks, etc.

In order to deliver on my promise to the BASIC users regarding the song-playing USR function, I must first lead the assembly language fanatics through a short intro to the Atari's interrupt system. As far as I know, the Atari is the only low-end personal computer that gives you such complete access to a fully-integrated, usable interrupt system. The Atari OS is structured to take advantage of several of these interrupts; and, more importantly, the user is invited to gain full or partial control of most interrupt routines. This despite the fact that Atari's interrupt service routines are in ROM.

The 6502 microprocessor supports two types of interrupts: NMI (Non-Maskable Interrupt) and

IRQ (Interrupt ReQuest). A bit in the CPU status byte controls whether IRQ's will generate interrupts, but if an NMI signal is presented to it the 6502 will always call in interrupt service routine. Atari, however, allows the user to prevent NMI's from reaching the CPU (except for the RESET button), thus giving even greater control. Once again, I must refer you to the Atari Technical Manual for full details, but herewith is a summary of the available interrupts.

Table 1. Available Interrupts

Type	Description
NMI	Reset Button (the only uncontrollable interrupt)
NMI	Display List Interrupt
NMI	Vertical Blank Interrupt (60 times per second)
IRQ	BREAK key
IRQ	any other key
IRQ	Serial Input (for SIO communication with disk, etc.)
IRQ	Serial Output (ditto)
IRQ	Serial Transmission Completed (ditto)
IRQ	Timer #4
IRQ	Timer #2
IRQ	Timer #1
IRQ	6520 parallel port "A"
IRQ	6520 parallel port "B"
IRQ	BRK instruction encountered (internal to 6502)

Each of the available interrupts, except the Reset Button and the BREAK key (and Timer #4 on all except newest machines), has a vector (two byte pointer) through RAM. To take control of an interrupt, simply put the address of your routine in the vector, and OS will call you instead of the default routine. The only exception is the Vertical Blank Interrupt, which is handled slightly differently and is the real subject of this article.

The Vertical Blank Interrupt (VBI) is really the key to many of Atari's unique features. It occurs 60 times per second, at the bottom of each scan of the TV screen, and is used by the OS ROMs to do all sorts of things. First, and perhaps most obvious, it drives the three-byte clock at locations \$12,\$13,\$14 (18,19,20 decimal) as well as several other usable event timers (e.g., serial bus timeout), most of which are accessible to the user. Second, and most useful, it allows changes to the graphics-related hardware at a time when nothing is being displayed on the screen: it moves all the "shadow" locations (see the technical manual) to their corresponding hardware ports.

Of necessity, then, the user would not normally want to interfere with the operations of the VBI routines. But, once again, the Atari software design team thought ahead: they provided not one, but two, VBI vectors. Thus, upon receipt of a VBI request, the ROM code first calls the routine pointed

to by vector VVBLKI (at \$0222) and then calls via the vector VVBLKD (at \$0224). The 'I' and 'D' stand for "Immediate" and "Deferred," respectively.

Normally, the user routine would not replace the vector at VVBLKI. Thus the Atari ROM code can update its clocks and move its "shadow" registers in confidence that it will finish its job before the screen starts displaying the next TV frame.

The user may replace VVBLKD to cause his routine to execute directly after the Atari system code.

Some cautions are in order:

- (1) Disaster will strike if your VBI routine is not done before the next VBI occurs. If you simply need to synchronize your routine to a vertical blank, just wait for the system clock to tick before starting (see the label WAITVB in this month's example program).
- (2) As with most Atari vectors, the safest way to use these is to move them somewhere in your own data area, replace them with your pointer, and have your code finish up by jumping back via the original Atari routine. This is particularly important to do with interrupt handlers, else the interrupt system may not be properly reset.

Finally, let me note that you may, if you really have to, steal the entire VBI processing for yourself. This is not necessarily bad (especially if you are writing a dedicated game, etc.), but be forewarned that you will have to worry about shadow registers, etc., yourself. There is a lot more to this subject, including what Atari refers to as time-critical I/O, but for most purposes you should be able to work within the rules I have outlined.

A Real, Live Example

The example program this month is designed to be used via USR from BASIC, but there is a simplified entry point from assembly language. You could lift this program as is and plunk it into any assembled game, etc. The idea behind the program is simple: a routine is passed a sequence of bytes which are interpreted to be commands to the sound genera-

tors of the Atari hardware. The routine examines the bytes and performs the requests. One of the available requests is to "play" sound(s) for a specified length of time; upon encountering this request, the routine waits the appropriate time before processing the next byte. Simple.

Except that this routine will operate (invisible to a running BASIC program) merrily playing

Main Assembly Listing

```

0000      1000      .PAGE "      equates, origins, etc."
0010      1010      ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
0020      1020      ;
0030      1030      ; PLAYIT -- a demonstration of performing
0040      1040      ;         clocked, interrupt-driven
0050      1050      ;         tasks under Atari OS.
0060      1060      ;
0070      1070      ; Written by Bill Wilkinson
0080      1080      ;         for March, 1982, COMPUTE!
0090      1090      ;
0100      1100      ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
0110      1110      ;
0600      1120      ORIGIN =    $0600
0000      1130      *=    ORIGIN
0140      1140      ;
00FF      1150      LOW    =    $FF
0100      1160      HIGH   =    $100
0170      1170      ;
D200      1180      AUDF1   =    $D200      ; Frequency, audio channel 1 (sound
0190      1190      AUDC1   =    $D201      ; Channel 1 control & volume
0224      1200      ;
0224      1210      VVBLKD  =    $0224      ; Delayed Vertical Blank routine
0220      1220      ;
0014      1230      CLOCKLSB =    $14      ; the system clock, LSB of 3
0240      1240      ;
00CE      1250      PLAYADDR =    $00CE      ; 2 byte pointer in safe place
0260      1260      ;
0270      1270      ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
0280      1280      ;
0290      1290      ; Equates for our private sound commands
0300      1300      ;
00FF      1310      CMDR    =    255      ; Repeat
00FE      1320      CMDS    =    254      ; Stop sound (keep routine going)
00FD      1330      CMON    =    253      ; Number of voices
00FC      1340      CMTV    =    252      ; set Tone and Volume
0000      1350      CMDE    =    0        ; End (but sound not turned off)
0360      1360      ;

0600      1370      .PAGE "      install our PLAYIT routine "
0380      1380      ;
0390      1390      ; INSTALL is the entry point called from BASIC
0400      1400      ;
0410      1410      ; The BASIC program calls us via
0420      1420      ;     USR( INSTALL, ADR(playit-command-string) )
0430      1430      ;
0440      1440      ; The routine may be called from
0450      1450      ;     assembly language at INSTALL1
0460      1460      ;     by placing the address of the
0470      1470      ;     command string in A,Y (LSB,MSB)
0480      1480      ;
0490      1490      INSTALL
0600      1500      PLA      ; BASIC tells us how many parameters
0601      1510      CMP      #1      ; better just have one!
0603      1520      GOOF    BNE    GOOF      ; else only RESET will get him out!
0605      1530      PLA      ;
0606      1540      TAY      ; MSB to Y register
0607      1550      PLA      ; LSB to A register
0560      1560      ;
0608      1570      INSTALL1 =    *        ; assembly language entry point
0580      1580      ;
0590      1590      ; first, we wait for a vertical blank
0600      1600      ; ...to ensure we don't get a VBLANK
0610      1610      ;     interrupt while we are working!
0620      1620      ;
0608      1630      A614      LDX    CLOCKLSB
0640      1640      WAITVB
060A      1650      CPX      CLOCKLSB      ; has clock ticked?
060C      1660      BEQ      WAITVB      ; no...keep waiting
0670      1670      ;
0680      1680      ; OKAY TO PROCEED
0690      1690      ;
060E      1700      STA      PLAYADDR      ; we preempted a zero page spot

```

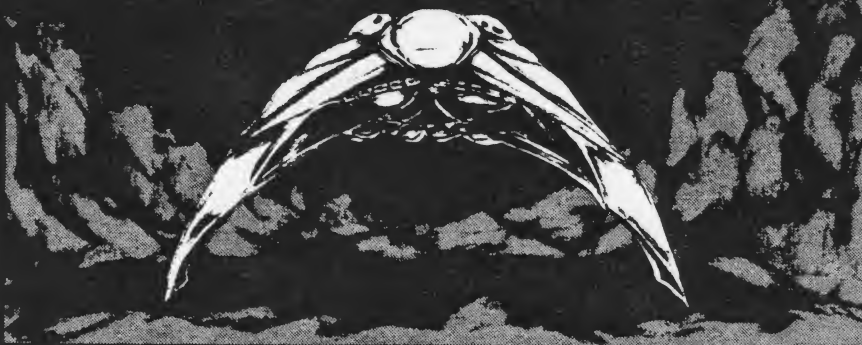

ong while BASIC continues
at it is doing. To accomplish
this, we have hooked into
VVBLKD (as described above).
The user specifies the note dura-
tion as a number of "jiffies" (60ths
of a second), and we let the VBI
count down the duration for us.

The commands are imbedded
in a string of bytes passed to the
routine. Playit recognizes six
command types, as shown in
Table 2. Playit is not particularly
sophisticated. For example, all
voices must play sounds for the
same duration and, when chang-

Table 2. Playit Command Codes

Byte value	Name	Description
255 (\$FF)	CMDR	Repeat the entire sound command string
254 (\$FE)	CMDS	Stop all sounds (do not end command string)
253 (\$FD)	CMDN	Number of voices is specified in next byte (0-4)
252 (\$FC)	CMDTV	Specify Tone and Volume (as in SOUND 0,freq, TONE,VOLUME). Must be followed by 0-4 bytes (one per each voice as specified by CMDN), each of which specifies a Tone/Volume for one channel.
0 (\$00)	CMDE	End command, unhook from VVBLKD. Does not turn off sound, so is usually preceded by CMDS.
any other	---	Any other value is assumed to be a duration, given in 'jiffies' (60ths of a second). Must be followed by 0-4 bytes (one per voice as specified by CMDN), each of which specifies the frequency of the sound for one channel (as in SOUND 0,FREQ, tone, volume).

SPRING SALE



CAVERNS OF MARS

ATARI APX

The surface of Mars is barren and rubble strewn, but beneath it lies a challenge only the brave and skillful dare undertake. Deep within the Red Planet lies the nerve center of the Martian's stronghold, protected by layer upon layer of the most ingenious defenses the crafty Martians can contrive. Your mission is to take on those legendary—and some say impregnable—defenses and to penetrate to the heart of the CAVERNS OF MARS! Player/Missile graphics, high-resolution color, and fine scrolling routines have been combined to create one of the most addicting games of all times.

Cat. No. 3452 16K, cass, joystick

\$29.95

Cat. No. 3453 24K, disk, joystick

\$29.95

48K

ATARI 400 OWNERS

48K

Now Atari 400 owners can solve the nagging problem of "you need more memory". The 48K RAMBOARD from Intec Peripherals comes completely assembled and tested and is fully documented for installation in the Atari 400. All contacts are gold plated for maximum reliability.

Cat. No. 3474

\$285.00

VERBATIM DATALIFE DISKETTES

Verbatim Datalife series has become synonymous with quality. If you have an ATARI, APPLE II, TRS-80, or any computer which uses soft-sector diskettes, you can take advantage of this low HW price.

Cat. No. 1147

\$28.00



YAHTMAN COMPUTER CORE SOFTWARE

YAHTMAN is a computerized version of a dice game which has been popular with young and old for years. Up to four people, school age and up, can compete for hours of enjoyment. YAHTMAN takes advantage of the Atari computer's graphics and can be played on both the Atari 400 and 800.

Cat. No. 3405 16K, cass, joystick

\$19.95

TT#5 PLAYER/MISSILE GRAPHICS

SANTA CRUZ SOFTWARE

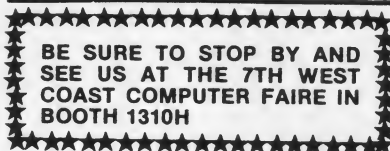
This is what it's all about. P/M Graphics is what set the Atari computer a cut above the rest when it comes to graphics. Learn how to create a simple shape called a player and you're on your way. This tutorial is loaded with 25 examples to create programs ranging from a complete business application to a small game.

Cat. No. 3400 32K, cass

\$29.95

Cat. No. 3401 32K, disk

\$29.95



HOW TO ORDER

Write or phone. Pay by check, M/C, VISA, or COD (add \$1.50 for COD). Offer expires Mar. 31, 1982. Mention this ad and we pay shipping (UPS ground only). (800) 423-5387. HW Electronics. 19511 Business Center Dr. Dept. G3 Northridge, CA 91324 (213) 886-9200

WHEN IN SOUTHERN CALIFORNIA,
VISIT OUR RETAIL STORES

HW ELECTRONICS

19511 Business Center Dr.
Northridge, CA 91324

2301 Artesia Blvd.
Redondo Beach, CA 90277

lines, if you DIMension more variables, etc., the string may move and Playit would start playing random sounds.

The commands have simply been entered into the program via DATA statements starting at line 9000. Those of you who go to the trouble to enter all this will, I hope, be pleasantly surprised by the sounds generated by lines 9400-9418. You will probably be dismayed, however, at the idea of putting in such a complex sound yourself. That is why I encourage someone to come up with a better "Music Compiler" along these same lines.

In any case, I invite you to compose your own music or sounds to be put into this system. Generally, I wrote a sound in BASIC to test it before committing it to DATA statements. For example, the "CHOO-CHOO" sound evolved from this BASIC line:

```
FOR V=15 TO 0 STEP -1 : SOUND
  V,V,O,V : NEXT V
```

The above sounds like an explosion, but if you slow it down a little and repeat it regularly you can train it as you wish. On to the short subjects.

HexDec

If you have already peeked at the listing of Playit From BASIC, you may have noted an unusual looking hexadecimal to decimal conversion routine. In fact, I herewith present you with a "one-liner" HexDec program:

```
1 DIM H$(23), N$(9): H$ = ",ABCDEF
  GHI!!!!!! JKLMNO": IN: N$: F.I =
  ITOLEN(N$): N = N*16 + ASC(H$(ASC
  (N$(I))-47))
:N.I?: N: RUN
```

The underlined characters are control characters (control-comma is the heart, etc.). The abbreviations are necessary to get it to fit on one line. To see how it works, figure out what happens when you input "9A". Recall that ASC("9") is 57 and ASC("A") is



CRYPTS OF TERROR

Beware as you enter the Crypts Of Terror. No one has survived this horror. Only your unrelenting nerve and determination will drive you deeper into the unknown.

Find what lurks in these ancient crypts!!

At last we have found an adventure with full graphics, sound and intrigue for your ATARI 400/800 computer.

• CRYPTS OF TERROR is the first adventure game that was completely designed for the Atari computers only. The graphics are the finest available using the full potential of the Atari.



Atari 800/400 16K requires joysticks.

Payment: Personal Checks – allow three weeks for check to clear.

American Express, VISA, MasterCard – include all numbers on card. Please include phone number with all orders.

Orders from USA \$29.95 (US funds)

Orders from Canada \$39.95 (Canadian funds)

Plus \$2.00 for shipping.

Ontario residents add 7% R.S.T.

Check your local computer dealer for Crypts Of Terror.

Dealer inquiries encouraged.

**INHOME
SOFTWARE**

PH. 1-416-961-2760

1560 Yonge St.
P.O. Box 10
Toronto
Ontario Canada
M4T 1Z7

65. 57-47 is 10 and 65-47 is 18. Look at the 10th and 18th characters in H\$. What is ASC("control-I")? ASC("control-J")?

You can avoid the control characters by adding the -64 shown in Playit From BASIC. Simple.

DecHex

This isn't really pertinent, but while we are on the subject of one-liners:

```
1DIMH$(16):H$="0123456789ABCDEF":IN:N:M=4096:F.I=1TO4:J=INT(N/M):H$(J+1);:N=N-M*J:M=M/16:N.N?:RUN
```

The USR And ADR Functions

Even though the methods of using the USR function are fairly thoroughly covered in the *Atari BASIC Reference Manual*, I find that many users are not fully aware of the real power of this function. Recall that the general syntax of this function is:

USR(addr [,expr [,expr ...]])

In other words, in addition to giving BASIC an address to call, you may pass *any number* of expressions to the assembly language routine. BASIC converts each expression to a 16-bit integer, pushes the result on the CPU stack, and cleans up by pushing on a single byte which tells the number of such expressions it pushed. (The address, which may itself be an expression, is *not* pushed and is not counted by that single byte.)

So what can we pass to assembly language? Obviously, numbers in the range of 0 to 65535. But what about characters? Conceive of

USR(addr, ASC("T"), expr),

where the "T" might be used as a mnemonic command to tell the routine which of several functions is desired. How about strings of characters? Recall that the three essential ingredients defining a

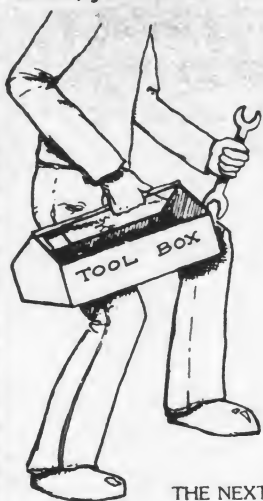
```
067B 4C4106 2550      JMP  SAM
                2560 ;
                2570 ;;;;;;;;;;;;;;
                2580 ;
                2590 ; set tone and volume
                2600 ;
                2610 DDTV
067E AEC606 2620      LDX  NUMVCS
0681 30BE   2630      BMI  SAM      ; no voices to set
                2640 TVLP
0683 20B706 2650      JSR  GETCMD   ; get next byte
0686 9D01D2 2660      STA  AUDC1,X  ; treat as t&v command
0689 CA     2670      DEX
068A CA     2680      DEX      ; more voices?
068B 10F6   2690      BPL  TVLP    ; yes
068D 30B2   2700      BMI  SAM      ; no
                2710 ;
                2720 ;;;;;;;;;;;;;;
                2730 ;
                2740 ; STOP the sound (by clearing all sound regs)
                2750 ;
                2760 DOSTOP
068F A207   2770      LDX  #7
0691 A900   2780      LDA  #0
                2790 STOPLP
0693 9D00D2 2800      STA  AUDF1,X  ;freq and vol to zero
0696 CA     2810      DEX
0697 10FA   2820      BPL  STOPLP
0699 30A6   2830      BMI  SAM      ; sound stops, pgm keeps going
                2840 ;
                2850 ;;;;;;;;;;;;;;
                2860 ;
                2870 ; END the processing (but doesn't stop sound)
                2880 ;
                2890 DOEND
069B ADC406 2900      LDA  SAVEVBLK
069E BD2402 2910      STA  VVBLKD   ; restore system ptr
06A1 ADC506 2920      LDA  SAVEVBLK+1
06A4 BD2502 2930      STA  VVBLKD+1 ; and, to OS, we aren't here
06A7 6CC406 2940      JMP  (SAVEVBLK) ; one last time
                2950 ;
                2960 ;;;;;;;;;;;;;;
                2970 ;
                2980 ; repeat the same stuff again
                2990 ;
                3000 DORPT
06AA ADC206 3010      LDA  REPEAT
06AD 85CE   3020      STA  PLAYADDR
06AF ADC306 3030      LDA  REPEAT+1
06B2 85CF   3040      STA  PLAYADDR+1 ; just reset the address
06B4 4C4106 3050      JMP  SAM      ; and try it again

06B7       3060      .PAGE "      the GETCMD subroutine"
                3070 ;
                3080 ; simply gets next byte from
                3090 ; command string
                3100 ;
                3110 GETCMD
06B7 A000   3120      LDY  #0
06B9 B1CE   3130      LDA  (PLAYADDR),Y ; get the byte
06BB E6CE   3140      INC  PLAYADDR  ; bump LSB of pointer
06BD D002   3150      BNE  GCEXIT   ; done
06BF E6CF   3160      INC  PLAYADDR+1 ; and the MSB
                3170 GCEXIT
06C1 60     3180      RTS
                3190 ;

06C2       3200      .PAGE "      ram usage"
                3210 ;
06C2 0000   3220 REPEAT .WORD 0      ; in case we hear it again
06C4 0000   3230 SAVEVBLK .WORD 0    ; so we can jmp indirect
06C6 00     3240 NUMVCS .BYTE 0      ; controls TVLP and FREQLP
06C7 00     3250 DURATION .BYTE 0    ; how long we hold a sound
                3260 ;
                3270 ;
                3280 .END

=0600 ORIGIN      =00FF LOW      =0100 HIGH      =D200 AUDF1
=D201 AUDC1      =0224 VVBLKD   =0014 CLOCKLSB  =00CE PLAYADDR
=00FF CMDR      =00FE CMDS     =00FD CMDN      =00FC CMDTV
=0000 CMDE      =0600 INSTALL  =0603 GOOF      =0608 INSTALL1
060A WAITVB     06C2 REPEAT    063C PLAYIT     0626 NOWINSTALL
0636 INSTALLED  06C4 SAVEVBLK   06C7 DURATION   066A EXIT
0641 SAM        06B7 GETCMD    069B DOEND      06AA DORPT
068F DOSTOP     067E DDTV      066D DONUM      065B DODURATION
06C6 NUMVCS     0660 FREQLP    0678 NUMOK      06B3 TVLP
0693 STOPLP     06C1 GCEXIT
```


Imagine being able to print the letter "A" and get a multi-color space ship. Using THE NEXT STEP and a minimum of programming effort, you can do it in no time at all.



THE NEXT STEP contains well-written, easy-to-use documentation with simple BASIC programming examples that show you how THE NEXT STEP can help develop colorful graphic displays. Graphics you never thought possible until now.

THE NEXT STEP is a user friendly, menu driven graphics tool kit that allows you to create new character sets or redefine characters to make shapes for use with your basic or machine language programs. THE NEXT STEP allows you to save these "new" characters on disk for future use.

THE NEXT STEP is perfect for use on shapes for animation and features a joystick controlled color menu to make your graphics come alive. THE NEXT STEP even generates its own code to help you incorporate new characters and shapes into your programs.

THE NEXT STEP allows you to see your shapes as you make them. Now you can determine ahead of time how characters will interact with one another when creating shapes for Character Set or Player-Missile Graphics.

THE NEXT STEP helps you to mix any of ATARI's 14 graphics modes in the same display. THE NEXT STEP is a perfect graphics utility for the BASIC or machine language programmer—novice and professional alike.

THE NEXT STEP features full joystick control for ease-of-use and quick editing.

THE NEXT STEP runs on any 32K ATARI 400/800 with a disk drive and is available for \$39.95 at your local computer store or order direct from

Q-N-LINE systems

THE NEXT STEP

VISA, MASTERCARD, CHECK, C.O.D.

Add \$1.00 for Shipping

36575 MUDGE RANCH ROAD • COARSEGOLD, CA 93614 • 209-683-6858

tring in Atari BASIC are its DIMension, LENGth, nd address. Since your program presumably DIMensioned the string, you know that value and ay pass it as an expression. And the address nd length are available from the ADR and LEN unctions!

Would you like your assembly language routine o modify your string, affecting its length? Try omething like this:

```
DIM XX$(XXDIM)
XX$(USR(addr,ADR(XX$),XXDIM)+1)=" "
```

Recall that the USR function may return any 16-bit alue to the BASIC program, which is automatically onverted to floating point as needed. Assume that his USR routine puts something in the XX\$ string nd returns the number of characters it put in. The above will then set the LENGth of XX\$ properly or use by other BASIC statements and functions.

Finally, there is floating point. How about riting a matrix inversion program? If we are imited to passing 16-bit integers, how do we pass a floating point number via USR? Simple: we pass he address of the number, just as we do with a string. And how do we get the address of a number, the ADR function only works with strings? Like this:

```
DIM FF$(1),FF(dim1,dim2)
JUNK=USR(addr,ADR(FF$)+1,dim1,dim2)
```

A little published fact about Atari BASIC is that DIMensioning of both strings and arrays proceeds in an orderly fashion according to the DIM statements encountered. And you are guaranteed that the order you DIM strings and arrays is the order they will occur in memory! So, by DIMensioning that one-byte string, FF\$, directly before the DIMension of the array, FF(), we *know* that the address of the array is one greater than the address of the string. Thus we can pass all the pertinent information about the array (its address and dimensions) to our assembly language routine. Incidentally, if you don't want to waste a one-byte string for this purpose, there is no reason FF\$ can't be any DIMension you need: just adjust the '+ 1' to reflect the actual DIM you use.

One last note on this subject: the fact that you can predict the memory order of strings and arrays has fascinating possibilities in regards to record structures, etc. But (and how many times have you read this from me) that's a topic for another article.

Program 1.

```
10 AUDCTL=53768:DBL=120
20 AUDF1=53760:AUDC1=53761
30 SOUND 1,10,10,15:SOUND 3,10,10,15
40 POKE AUDC1,0:POKE AUDC1+4,0
50 POKE AUDCTL,DBL
60 FOR J=10 TO 15:POKE AUDF1+2,J:POKE AUDF1+6,20-J
```

```

70 FOR I=0 TO 255:POKE AUDF1,I:POKE AUDF1+4
,255-I:NEXT I
80 NEXT J

```

...VERY SMOOTH GLIDES...

Program 2.

```

10 AUDCTL=53768:DBL=120
12 OSC=1789790/2
20 AUDF1=53760:AUDC1=53761
30 SOUND 1,10,10,0
40 POKE AUDC1,0:POKE AUDC1+4,0
50 POKE AUDCTL,DBL
60 P2=2^(1/12)
70 NTE=16:REM C IN THE REAL BASS
80 FOR I=1 TO 109
90 FREQ=INT(OSC/NTE-7+0.5):F0=INT(FREQ/256)
92 F1=FREQ-256*F0
100 POKE AUDF1,F1:POKE AUDF1+2,F0
102 POKE AUDC1+2,175
103 PRINT "NOW PLAYING ";INT(NTE+0.5);" HZ"
105 FOR J=1 TO 100:NEXT J
110 NTE=NTE*P2
120 NEXT I
130 GOTO 70

```

...9 OCTAVE CHROMATIC SCALE...

Playit From BASIC

```

1000 REM *****
1020 REM *
1040 REM * PLAYIT FROM BASIC, SAM
1060 REM *
1080 REM * This routine is a simple
1100 REM * sound "compiler", which
1120 REM * takes DATA statements and
1140 REM * converts them into command
1160 REM * strings suitable for use by
1180 REM * the interrupt-driven PLAYIT
1200 REM * routine.
1220 REM *
1240 REM *
1260 REM * Written by Bill Wilkinson
1280 REM *
1300 REM * for March, 1982, COMPUTE!
1320 REM *
1340 REM *****
1360 REM
1380 REM First. constants, routine addresses, etc.
1400 REM
1420 DIM HX$(2),CMD$(11),PLAY$(1000),HEX$(23),TYPE$(1),
PLAYIT$(1000)
1440 HEX$="@ABCDEFGHIJ!!!!!!JKLMNO"
1460 DOCMD=2300:LOOP=1800:HEXDEC=2600
1480 AGAIN=1700:EXITLOOP=2100
1500 PLAYIT=6*256:REM or wherever you put the routine
1520 REM
1530 SOUND 0,0,0,0:REM needed to initialize properly
1540 REM The command equates...
1560 REM notice that these match the
1580 REM assembly language routine
1600 CMDR=255:CMDS=254:CMDN=253:CMDTV=252:CMDE=0
1620 REM
1640 REM *****
1660 REM
1680 REM This is the AGAIN of
1700 REM PLAY IT AGAIN, ATARI
1720 REM
1730 PRINT " <processing...please wait>"
1740 PLAY$="":PLAY=0
1760 REM
1780 REM This is LOOP
1800 PLAY=PLAY+1:REM to next cmd byte
1820 READ CMD$:REM a bunch of commands
1840 REM
1860 TYPE$=CMD$:REM use the command character
1880 IF TYPE$="R" THEN PLAY$(PLAY)=CHR$(CMDR):GOTO EXIT
LOOP

```

```

1900 IF TYPE$="S" THEN PLAY$(PLAY)=CHR$(CMDS):GOTO LOOP
1920 IF TYPE$="N" THEN NUMVCS=1:CMD=CMDN:GOSUB DOCMD:NUM
VCS=DEC:GOTO LOOP
1940 IF TYPE$="T" THEN CMD=CMDTV:GOSUB DOCMD:GOTO LOOP
1960 IF TYPE$="E" THEN PLAY$(PLAY)=CHR$(CMDE):GOTO EXIT
LOOP
1980 REM *** IF TO HERE, ASSUME DURATION & FREQ ***
2000 HX$=CMD$:GOSUB HEXDEC:CMD=DEC:REM command is
duration
2020 CMD$=CMD$(2):REM to fool DOCMD
2040 GOSUB DOCMD:GOTO LOOP
2060 REM
2080 REM exitloop
2100 REM
2120 REM do the sound playing
2140 REM
2150 PLAYIT$=PLAY$:REM else we alter what we are playing
2160 JUNK=USR(PLAYIT,ADR(PLAYIT$))
2180 REM
2200 PRINT "HIT RETURN FOR NEXT SOUND ";:INPUT TYPE$
2220 GOTO AGAIN
2240 REM
2260 REM
2280 REM *****
2300 REM THE SUBROUTINES
2320 REM
2340 REM first, DOCMD
2360 REM
2380 PLAY$(PLAY)=CHR$(CMD):REM The command byte
2400 IF NUMVCS=0 THEN RETURN
2420 REM we process NUMVCS bytes
2440 FOR I=2 TO NUMVCS+NUMVCS STEP 2
2460 HX$=CMD$(I):GOSUB HEXDEC:REM convert the byte
2480 PLAY=PLAY+1:PLAY$(PLAY)=CHR$(DEC):REM and stuff it
away
2500 NEXT I
2520 RETURN
2540 REM
2560 REM .....
2580 REM *****
2600 REM and now HEXDEC
2620 REM
2640 DEC=0:REM our accumulator
2660 FOR L=1 TO LEN(HX$)
2680 DEC=DEC*16+ASC(HEX$(ASC(HX$(L))-47))-64
2700 NEXT L
2720 RETURN
8999 REM ...a siren-like sound...
9000 DATA N01,TCF,1408,1412,R
9099 REM ...a fanfare of sorts...
9100 DATA S,N01,TA2,30F3
9102 DATA N02,TA3A3,30F3C1
9104 DATA N03,TA4A4A4,30F3C1A1
9106 DATA N04,TA5A5A5A5,60F3C1A17A
9108 DATA T00000000
9110 DATA N00,C0,R
9199 REM ...beeping off the seconds...
9200 DATA S,N01
9202 DATA TAE,0130
9204 DATA TAC,0130
9206 DATA TAA,0130
9208 DATA TAB,0130
9210 DATA TA6,0130
9212 DATA TA4,0130
9214 DATA TA2,0130
9216 DATA T00,3500
9218 DATA R
9299 REM ...choo-choo ??? ...
9300 DATA S,N01
9302 DATA T0E,010E
9304 DATA T0C,010C
9306 DATA T0A,010A
9308 DATA T0B,010B
9310 DATA T06,0106
9312 DATA T04,0104
9314 DATA T02,0102
9316 DATA T00,0300
9318 DATA R
9400 DATA S,N01,TAC
9402 DATA 3051,305B,3044,183C,182D,3035
9404 DATA ^3C,182D,3035,3044,303C,3051,305B
9406 DATA F04,TACA4A4AB
9408 DATA 30516C89A2,305B7990B6,30446C89A2
9410 DATA 183C4879B6,182D4879B6,3035485B07
9412 DATA 183C4879B6,182D58B686,30354458B9
9414 DATA 3044516CA2,38325179F3
9416 DATA 423C4858B6,5044586C89
9418 DATA S,N00,F0,R
9898 REM ...stop and end...to quit...
9999 DATA S,E

```

INSIGHT: Atari

Bill Wilkinson
Optimized Systems Software
Cupertino, CA

This month, I present a session on how to steal a system. Before all you kleptomaniacs rejoice, though, I should explain that I mean to show you how to take control of your Atari's software system when a user pushes the SYSTEM RESET button. This will, I hope, be useful to BASIC and assembly language programmers alike.

There will be more on the inner workings vs. outer appearances of Atari BASIC; and, as space permits, I will have my usual assortment of cute tricks and Did You Knows.

In a departure from the norm, I will review a product here. Since my company, Optimized Systems Software, both solicits and sells software for Atari (and Apple) computers, I do not think it would be fair for me to do software reviews. But, unless you and/or my dear editor object, I may, from time to time, discuss new and wondrous happenings in the world of Atari.

A Short Review

It generally strikes me as unfair for a magazine to carry a review of something it sells; but every other magazine does it, so I prevailed on **COMPUTE!** to let me review *COMPUTE!'s First Book of Atari*. I am doing so on the condition that the review must be run as I submit it. (Okay, okay, Richard...you can fix my punctuation.) I have to do a good review or they won't let me do it again (just kidding...I think).

First, let me say that I did not start reading **COMPUTE!** regularly until about December, 1980, so most of the material in the book was new to me. Boy was it new to me! Quite frankly, I had lulled myself into thinking that, until I started writing my column, the poor Atari user had no insight (an insight gag) into the workings of Atari BASIC, DOS, etc. *Not so!!* There was a lot of good stuff published in **COMPUTE!** during 1980.

I don't want to make this review sound like a whitewash, so let's get the bad stuff over with first. The first warning that needs to be given is that, in general, this is not a book for software hackers: there is little of interest to the assembly language programmer (but see below for some notable exceptions), and the person who has read *and understood* (!!) the technical manuals and, perhaps, *De Re Atari* won't find much he or she didn't know. However, for most people there is much useful material

here. There are some bloopers in the material presented, things which probably wouldn't get past the current, more Atari-sophisticated, editorial staff. There's a little duplication of material. And there's a lot of stuff that has been updated by better articles which have appeared (usually in **COMPUTE!**) in 1981 and 1982. Actually, my biggest complaint is in a reprinted article titled "The OUCH in Atari BASIC": the article states, *and* the editorial lead-in agrees (and the lead-in was written recently – Oh, for shame!), that keywords can't be used in variable names. Yet, the very next article in the book says that all keywords can be used as variable names! (Still not quite true – "NOT" is poison as the first three characters of a name, and a few keywords, such as "TO" and "STEP" can't be used as-is. Oh well, this was 1979 and 1980. And, come to think of it, even Atari's *BASIC Reference Manual* still says not to use keywords as names. Of course, it also says that substrings are limited to 99 characters, so maybe it's not a good reference point.)

OK. So much for the bad stuff. "Not possible!" you scream? Sorry, but it's true. I really don't have any dirt to sling. Oh, some of the little example programs might now be found in the Atari manuals, etc., but they aren't *bad*, just not of as much value as the rest of the book. And I wish I had the time and space to correct every little goof I found. (But I gotta tell you *one*: the order and size of variables and their names has *no* impact at all on the speed of an Atari BASIC program. Honest.) With those caveats in mind, we examine the value of the book.

And the book is of value. If you had to choose between losing your left pinkie (not quite up to the left thumb, anymore), the *First Book*, and *De Re Atari*, you should really think about how useful a little pinkie is. If you *must* choose between *De Re* and *First Book*, let your experience level be your guide: if you almost understand the Atari technical manuals, you are probably ready for *De Re*. If you are just learning to program, stick with *First Book*. If you're in the middle, better let the little pinkie go.

My own favorite pair of articles from the book are "Inside Atari BASIC" and "Atari Tape Data Files," both by Larry Isaacs. I am just now getting to the point where I am discussing things in "Insight: Atari" that Larry explored over a year ago (there will be overlap, hopefully to your benefit). Other articles worth mentioning include the following (an asterisk indicates something of interest to assembly language buffs):

"Printing to the Screen from Machine Language"*
(not because of what the presented program does as much as for some of the techniques it introduces)

"The Fluid Brush"
(Ditto. And its ideas have been much copied.)

"Player/Missile Graphics" *

(by Chris Crawford. What more need I say?)

"Adding a Voice Track to Atari Programs" *

(This one was even swiped! There are more sophisticated methods shown in *De Re*, but this is adequate for many purposes.)

"Atari Memory Locations" *

(Just a table. You need to read the *Technical Manual* and/or *De Re* first, but this will serve as a handy reminder.)

"Input/Output on the Atari"

(I hesitated on this one: you should ignore what it says about XIO! It's misleading. Read my Atari I/O series.)

You'll note that most of that stuff is kind of heavyweight. Well, that's what appeals to me and, I think, to a large portion of **COMPUTE!**'s Atari readership. However, there are several little goodies, usable by virtually anyone, which deserve honorable mention. No commentary on these: their names tell it all and you just have to try them to appreciate them:

"Reading the Atari Keyboard on the Fly"

"Al Baker's Programming Hints"

"Atari Sounds Tutorial"

"Using the Atari Console Switches"

"Atari Meets the Real World"

You may have your own favorites, but my criterion for a good article (or good magazine-published program) is that it teaches you something. Thus I rate type-it-in-and-run-it games relatively low. (There are remarkably few of them that appear in the book.)

In final summary, I have to say that, for \$12.95, you are unlikely to find this much (184 pages, including – can you believe it – a usable index) useful Atari material presented again (well...until the *Second Book*?). Real software hackers will find some of the material too elementary, but they are probably the only ones that will be disappointed.

Stealing A System

During my series on Atari I/O (**COMPUTE!** November, 1981, through March, 1982, issues 18 through 21), I mentioned (more than once) the "proper" way to add device drivers to OS. I summarize it here again:

1. Inspect the system MEMLO pointer (at \$2E7).
2. Load or relocate your routine at/to the current value of MEMLO.
3. Add the size of your routine to MEMLO.
4. Store the resultant value back in MEMLO.
5. If your routine is a device driver, connect

it to OS by adding its name and address to HATABS.

6. Fool OS so that steps three through five will be re-executed if SYSTEM RESET is hit.

In **COMPUTE!** (January, 1982, #20) we added the driver for the "M:" device by following steps one through five as above. We discussed step six briefly, but did not show how to implement it. This month, we will show how to fool OS. And, rather than repeating the lesson about adding device drivers, we will take this opportunity to show how to give Atari BASIC some measure of control over what happens on RESET.

In particular, we "steal" the system in a way that the user who hits RESET will cause a TRAP-able error in the running BASIC program. In other words, if you write your BASIC program in a way that TRAP (to a line number) is always active, you will be able to detect when your user hits the RESET key, but your program will not stop running, will not lose its variable values, and will be impacted in the minimum possible way.

Some cautions are in order (it seems like I always have to say that): *before* vectoring through RAM (and thus allowing our little trick) Atari's OS ROMs perform several actions when SYSTEM RESET is hit. If you need to know exactly what happens, try to get hold of the CIO listings (they are moderately readable); generally, the following lists all that matters except to those who would make their own cartridges:

1. The system resets any memory size pointers (MEMLO, MEMTOP, etc.).
2. Most hardware registers are reset to zero (\$D000-\$D0FF, \$D200-\$D4FF).
3. OS clears its own RAM (\$200-\$3FF, \$10-\$7F). Note that this zaps all IOCB's for all files.
4. All the ROM-based device drivers are initialized (via their own initialize routines).
5. CIO's initialization is called, which effectively marks all files as properly closed.
6. Screen margins, etc., are reset and the E: device is opened on file channel #0 (which is equivalent to GRAPHICS 0 from BASIC).
7. The file manager's initialization routine is called via an indirect call through location DOSINIT (\$0C).
8. If there are no cartridges installed, then DOS is invoked by an indirect jump through location DOSVEC (\$0A). If a cartridge is installed and wants control, though, OS goes to the cartridge instead of DOSVEC.

(NOTE: OS/A+ uses a variation on 7., above,

so don't bang your head against the wall trying what is written here with OS/A+. I will be glad to tell you of the differences if the manual is not clear enough.)

Program 1 takes into account not only all of the above, but also the requirements of Atari BASIC related to executing a pseudo-warmstart. I will not try to explain why the various JSRs and tests shown are needed; just take my word for it that they are indeed necessary (I found out the hard way). Actually, the part pertaining to stealing the DOSINIT vector is straightforward, as you may note, and changing MEMLO is trivial.

Once again, for space and time reasons, I have cheated with this program: I have assumed that my routine can load and execute at \$1F00 and can move MEMLO to \$2000. Those of you who want to do the whole thing might can follow the techniques I showed in **COMPUTE!** February, 1982, #21, for generating relocatable programs. Also, please note that the listing, as is, is designed to produce an AUTO-RUN.SYS file. You may need to do a little surgery to use it in other ways (e.g., remove the load-and-go vector at the end, JMP directly to the start of the BASIC cartridge, etc. — experiment).

The most important thing to note about this routine's implementation is how the address found in DOSINIT is moved into the JSR instruction (at the label RESET). Obviously, you could go look at the contents of DOSINIT and code the JSR directly, omitting the move of the address. And this will work as long as you use the same version of OS and DOS. But ... all too many Atari software developers fell into the trap of thinking that OS and DOS were immutable, only to have Atari announce DOS 2S and OS version B.

To Atari's credit, they have

CRUSH, CRUMBLE, AND CHOMP



AUTOMATED SIMULATIONS

Breathe fire, terrorize cities, snack on a horrified populace, and further develop your villainous personality in CRUSH, CRUMBLE, AND CHOMP. Take on the persona of any of six demonic beasts (even more for those who have a disk), select from four mouth-watering metropolises, choose one of five different objectives, and start wreaking havoc!

Cat. No. 3532 Atari, 32K, disk
Cat. No. 3536 Atari, 32K, cass

\$29.95
\$29.95

WIZARD AND THE PRINCESS

ON-LINE SYSTEMS

Assume the role of a poor but happy vagabond wandering through the land of Serenia and let the fun begin. As you enter a village, you notice a town crier. There's trouble afoot—the evil wizard Harlin has abducted the King's daughter Princess Priscilla. Half the kingdom is offered as a reward for her safe return and, what the heck, you could use the money. To reach Harlin's castle you must cross a vast desert and enter the mountains equipped with only the bare essentials. Oh, well—beggars can't be choosy!

Cat. No. 3556 Atari, 40K, disk, machine language

\$32.95

HOCKEY

GAMMA SOFTWARE

HOCKEY is a challenging high-speed video action game of skill for two, three, or four people. Written in machine language, HOCKEY produces realistic action involving two four-man teams. An offensive player can carry the puck, pass, and shoot. A defensive player can steal the puck and intercept passes. Even the goalie gets a piece of the action. An advanced feature of this program is the inclusion of "smart" players who perform automatically. In case of a tie after regulation time, there is even sudden-death overtime! High resolution color, sound, multiple-play options, and real-time action make HOCKEY an exciting game for your Atari 400 or 800 computer.

Cat. No. 3588 Atari, 16K, cass, joysticks (2+)
Cat. No. 3587 Atari, 16K, disk, joysticks (2+)

\$29.95
\$29.95

VERBATIM DATALIFE DISKETTES

The ideal diskette for Atari, Apple, TRS-80, or any soft-sector computer. Each diskette is rated for single or double density and will work on 40-track disk drives. A hub ring is also installed for increased reliability.

Cat. No. 1147
(PRICE INCLUDES SHIPPING)

\$28.00/box of 10



EPROM BOARD for Atari

EASTERN HOUSE SOFTWARE

Now Atari owners can implement their utilities or applications in firmware. Designed to install in the cartridge slot of the Atari computer, this EPROM board can contain up to 8K of program code in either 2716's or 2732's.

Cat. No. 3477

\$19.95

HOW TO ORDER

Write or phone. Pay by check, M/C, VISA, or COD (add \$1.50 for COD).
(800) 423-5387 (213) 886-9200
Offer expires Apr. 30, 1982

Mention this ad and we pay shipping (UPS ground only).
HW Electronics 19511 Business Center Dr. Dept. G4
Northridge, CA 91324

WHEN IN SOUTHERN CALIFORNIA, VISIT OUR RETAIL STORES

HW ELECTRONICS

19511 Business Center Dr.
Northridge, CA 91324

2301 Artesia Blvd.
Redondo Beach, CA 90277



CRYPTS OF TERROR

Beware as you enter the Crypts Of Terror. No one has survived this horror. Only your unrelenting nerve and determination will drive you deeper into the unknown.

Find what lurks in these ancient crypts!!

At last we have found an adventure with full graphics, sound and intrigue for your ATARI 400/800 computer.

• CRYPTS OF TERROR is the first adventure game that was completely designed for the Atari computers only. The graphics are the finest available using the full potential of the Atari.



Atari 800/400 16K requires joysticks.
Payment: Personal Checks – allow three weeks for check to clear.

American Express, VISA, MasterCard – include all numbers on card. Please include phone number with all orders:
Orders from USA \$29.95 (US funds)
Orders from Canada \$39.95 (Canadian funds)
Plus \$2.00 for shipping.

Ontario residents add 7% R.S.T.
Check your local computer dealer for Crypts Of Terror.
Dealer inquiries encouraged.

**INHOME
SOFTWARE**

PH. 1-416-961-2760

1560 Yonge St.
P.O. Box 10
Toronto
Ontario Canada
M4T 1Z7

carefully documented which locations, vectors, etc., are guaranteed to remain unchanged. If you write your code properly, you should never have to change it. Incidentally, another example of this same concept appeared in my last article: the vector from VVBLKD was preserved, rather than simply jumping to the routines in ROM.

Enough preaching: investigate the listing. But I leave you with one last freebee. If you change three lines of code in the listing (lines 1950 to 1970) to the following two lines, you will cause BASIC to reRUN the program currently in memory, rather than causing an error TRAP.

1950 JSR \$B755
1960 JMP \$A962

The following short BASIC program illustrates the use of our stolen pointer:

```
10 TRAP 100
20 PRINT "LOOPING AT
   LINE 20"
30 GOTO 20
40 STOP : REM
```

can't get here from there...
or anywhere

```
100 IF PEEK(195)<>255 THEN
   PRINT "HOW? WRONG
   ERROR CODE!":STOP
110 PRINT "RESET KEY WAS
   HIT...I WILL START
   AGAIN"
120 RUN
```

Note that you *can't* get out of this program with the RESET key! Now, if you also trap the BREAK key, the user is truly locked in your program. If you have BASIC A+, of course, you can trap the BREAK key via SET 0. If not, then refer to the listing titled "Idiot-Proofing the Keyboard" in *De Re Atari*. (Summary of that listing: since the BREAK key is one of the two IRQ's not vectored through RAM, you must change the system master IRQ vector to point to your own routine. In your

routine, you check for and ignore BREAK key IRQ's and pass other IRQ's on unchanged. Not trivial, perhaps, but certainly less complicated than what we have done above.)

Inside Atari BASIC, Part 3: Enhancements

After skipping last month, we return to our discussion of the hows and whys of Atari BASIC. Recall that in **COMPUTE!** February, 1981, #21, we discussed how BASIC checks your entered line for correct syntax and produces a tokenized result. Let us begin this month with a discussion of how BASIC executes (RUNs) a program.

First, note that if you enter a direct line (one without a line number), BASIC arbitrarily assigns it to line number 32768 and then pretends that it is like any ordinary line. That means that even direct lines must go through the tokenizing and execution process. It also means that BASIC makes little or no distinction between statements (within a program) and commands (given directly); thus you can LIST or RUN or even CONTINUE from inside a BASIC program.

Whenever a line is finished executing, BASIC checks to see if the next line exists (it doesn't if a direct line was just executed) or if the next line has a line number greater than 32767 (i.e., if the line executed is the last one prior to the direct line). If either condition prevails, BASIC pretends to itself that it got an "END" statement and, presto, you are back staring at the "READY" prompt.

But let us assume that the direct command given was "RUN." The execution of a RUN statement simply causes all BASIC's pointers and flags to be reinitialized, including setting BASIC's "next line" pointer to the beginning of the program. Then RUN returns to what we call "Execution Control" which decides that it needs to start executing the next line...which conveniently is the first line.

So far, so good. But how does BASIC know what to do with the tokens? The answer is that it doesn't, really. Recall that there are two separate kinds of execution (as opposed to variable) tokens: statements and operators. Each of these has a table of two-byte pointers residing in BASIC's ROM. Execute Control simply picks up the next byte of the program, assumes that it is a statement token (incidentally, in the range of \$00-\$7F), and uses double its value as an index into the table of statement pointers. It uses the address thus found as an indirect jump and goes to the appropriate statement execution routine.

In a non-syntaxed BASIC (i.e., Microsoft), much of the preceding applies virtually unchanged. But, when the statement execution routine gets control in such a BASIC, it has no idea what the

next character or token in the program might be, so it must needs go through a set of checks to determine what is legal and what is not.

In Atari BASIC, though, the statement execution routine *knows* that the byte(s) that follow constitute legal syntax! So it need not waste time checking for legality. Since the bytes following the statement token may range from the non-existent (as in CONT, which has no following bytes) to the extremely complex (as in PRINT, in all its variations), each statement generally has responsibility for choosing what to do with these bytes.

With the exception of assignment-type operations (LET, READ, INPUT, etc.), file designators (PRINT #), and complex statements (FOR...TO...STEP), what follows the statement byte is generally a series of one or more expressions, separated by commas, equal signs, semicolons, etc. Thus it comes as no surprise that there is a major subroutine in Atari BASIC entitled "Execute Expression," which can evaluate virtually any numerical or string expression.

As a simple example, let us examine the mechanism of POKE. The syntax is properly "POKE <aexp>,<aexp>" (where <aexp> means Arithmetic EXpression). So POKE's statement execution routine simply calls Execute Expression for the first value, saves it away someplace safe, skips the comma (it *knows* the comma is there...the syntaxer said so!), calls Execute Expression for the second value, and stores the second into the memory location designated by the first. Now, in truth, POKE calls a variation on Execute Expression which is guaranteed to return a 16-bit (or 8-bit, as required) value; but the concept holds for most statements.

It is really beyond the scope of this article to try to explain the intricacies of Execute Expression. It will suffice to point out that it must worry about operator precedence ("*" before "+", etc.) and parentheses and subscripted variables and functions (SIN, RND, etc.) and more.

And that's about it. Except to note that when a statement is finished it usually simply returns to Execute Control, which checks for another statement on the same line and/or moves its pointer to the first statement of the next line.

Much of the point of this discussion has been to show why it is hard to fool BASIC into believing that it has a new statement to use. Even with the source, it is no easy task to make sure that the correct syntax for a new statement is entered into the syntax tables (which are actually a miniature language in their own right), the name tables, and the execution tables (to say nothing of writing the code to execute the statement). With Atari BASIC locked in ROM, the task is really impossible since BASIC makes use of no RAM-based pointers or

indirect jumps throughout this process.

So how can we add features to Atari BASIC? Several ways:

1. Try the USR function as suggested last month. This really is the simplest, most straightforward, most guaranteed-to-work.
2. Make your own special device handler (a la "M:" in **COMPUTE!**, January, 1981, issue #20). Open a channel to it (OPEN #1,...). PRINT something to it. When your driver gets control, it can actually go in and look at the BASIC tokens and decide what to do from there. Cryptic, but it works.
3. If you are interested in commands, as opposed to statements, you can intercept BASIC's call to "E:" (for the next input line) and examine the line yourself (presumably as does Monkey Wrench). This implies that you must check syntax, find variables, convert ASCII to floating point, etc., in your routine. Tedious, but obviously feasible and usable.

As you can, no doubt, tell, I am much in favor of method 1. It is by far the easiest to do and requires the least knowledge of BASIC's internals.

Is there yet another way? A month ago I would have said "no!" But, now, I have discovered a crack in the door. It is complicated, prone to programmer error, fairly inflexible, and of doubtful value for anyone but professional software developers. To explain it would take a couple of more columns, and I'm simply not willing to write that much on a topic of dubious value. If you feel you absolutely *must* know, write me (care of OSS). If enough people write, I *may* make up a pamphlet and sell it at an outrageous price. Are you sure you can't live with method 1?

Easy Horizontal Joysticks

If you have an application (a polite way of saying game) that needs a joystick that moves only horizontally (or only vertically, if you are willing to hold your joystick turned 90 degrees from "normal"), then have I got a trick for you! Try this program, with joystick number 0 plugged in:

```
10 PRINT PTRIG(0)-PTRIG(1),
20 FOR J=1 TO 50 : NEXT J
30 GOTO 10
```

Now push the joystick in all directions. Neat? Pushing it left gives you a value of -1 and right gives you +1. And, of course, you can use A = PTRIG(2)-PTRIG(3) to read joystick number 1, etc.

Why does it work? Because the paddle triggers happen to use the same pins on the connector that the horizontal switches in the joystick use. I discovered that by reading the technical manual; so, you see, there is probably still buried gold in those

books.

Unfortunately, no such happy coincidence exists for reading the vertical joystick switches. Incidentally, use of this trick does not affect STRIG in any way.

Dissonances

The algorithm Atari gives for figuring out what *actual* frequency will result from the divider **FREQ** in (for example) **SOUND 0,FREQ,tone,volume** is as follows:

$$\text{actual frequency} = \frac{63921}{\text{FREQ} + \text{FREQ} + 2}$$

This means that, at values for **FREQ** around 85 (the middle of the Atari's frequency range), the minimum actual frequency step is about 4 Hz. While adequate for solo parts, this kind of frequency resolution can really grate on your ears when there are three or four voices active. To illustrate the real meaning of this, try the following one-liner:

```
FOR F=255 TO 0 STEP -1 : SOUND 0,F,10,15 :
NEXT F
```

The resultant sound is a smooth glide until you get near the top end, when you begin to really hear the steps.

For those of you with a keen ear and/or a strong sense of music, cheer up. Atari, once again, gave us a solution. The entire Atari audio system is controlled by hardware register **AUDCTL** (\$D208). Normally, the audio channels are clocked by an oscillator running at 63921 Hz. But, the user may specify that channels zero and two (which Atari calls one and three in the *Technical Manual*... oh well) are to be clocked by a 1,789,790 Hz oscillator. If you change 63921 to 1789790 in the formula above and plug in 255 (the highest value) for **FREQ**, you will see that the *lowest* note thus playable is around 3000 Hz!

But we have yet another solution available via **AUDCTL**: instead of an 8-bit counter for a single audio channel, we use a pair of channels to produce a 16 bit counter. (Unfortunately, we then are limited to two sound channels.) The modified formula then becomes:

$$\text{actual frequency} = \frac{1789790}{\text{FREQ} + \text{FREQ} + 14}$$

Since **FREQ** now has values from 0 to 65535, it's obvious we have many more actual frequencies available to us. I present herewith two sample programs using this technique. Program 2 shows two voices doing very smooth glides. Program 3 shows a 9-octave chromatic scale (C, C#, D, D#, etc.). This compares to the 3-octave scale available

via the standard SOUND commands.

Finally, if you simply need lower notes than

you can get with SOUND, TRY POKEing AUDCTL to one. This has the effect of lowering *all* notes by

Program 1.

equates and commentary

```

0000      1000      .PAGE "equates and commentary"
          1010 ;
          1020 ; STEAL A RESET
          1030 ;
          1040 ; listing for COMPUTE! magazine, April, 1982
          1050 ;
          1060 ;
          1070 ; there are two parts to this listing:
          1080 ;   1. what happens when this is first loaded
          1090 ;       (which initializes everything)
          1100 ;   2. what happens when the user pushes
          1110 ;       SYSTEM RESET.
          1120 ;
          1130 ; Most of 1 is understandable. Most of 2
          1140 ; is magic. If it works, don't knock it.
          1150 ;
02E7      1160 MEMLO =    $2E7      ; BOTTOM OF USABLE MEM
          1170 ;
00FF      1180 LOW   =    $FF
0100      1190 HIGH  =    $100
          1200 ;
          1210 ; EQUATES INTO BASIC ROM
          1220 ;
BD72      1230 SETDZ  =    $BD72      ; ENSURE OUTPUT TO CONSOLE
0092      1240 MEOL  =    $92        ; FLAG: LINE MODIFIED
BD99      1250 FIXEOL =    $BD99      ; UNMODIFY
00B9      1260 ERRNUM =    $B9        ; AT LEAST BASIC THINKS SO
B940      1270 ERROR  =    $B940     ; HANDLE ERRORS
00BD      1280 TRAPFLAG = $BD
DA51      1290 INITBUF = $DA51      ; SAFETY
0011      1300 BRKFLAG = $11
BD41      1310 CLOSEALL = $BD41     ; close IOCBs 1-7
000C      1320 DOSINIT = $0C        ; see article
          1330 ;

```

SETUP THE RESET VECTOR

```

0000      1340      .PAGE "SETUP THE RESET VECTOR"
          1350 ;
          1360 ; We move the OS DOSINIT vector to point to ourselves
          1370 ;
          1380 ; ***** NOTE: change next line to suit!!! *****
0000      1390      *=    $1F00
          1400 CHANGEDOSINIT
1F00 A50C  1410      LDA  DOSINIT
1F02 8D231F 1420      STA  RESET+1
1F05 A50D  1430      LDA  DOSINIT+1 ; Self modifying code...nasty
1F07 8D241F 1440      STA  RESET+2
1F0A A922  1450      LDA  $RESET&LOW
1F0C 850C  1460      STA  DOSINIT

```

```

1F0E A91F    1470      LDA  #RESET/HIGH
1F10 850D    1480      STA  DOSINIT+1 ; We have changed the pointer
           1490 ;
           1500 ; Here we move MEMLO...
           1510 ; we arbitrarily use 256 bytes of space
           1520 ;
           1530 MOVEMEMLO
1F12 A900    1540      LDA  #RESETTOP&LOW
1F14 8DE702  1550      STA  MEMLO
1F17 A920    1560      LDA  #RESETTOP/HIGH
1F19 8DE802  1570      STA  MEMLO+1
1F1C 60      1580      RTS
           1590 ;
           1600 ; FROMBASIC is just a second entry
           1610 ; entry point into the initialization...
           1620 ; for initializing from BASIC
           1630 ;
           1640 FROMBASIC
1F1D 68      1650      PLA          ; get cnt of parms off stack
1F1E F0E0    1660      BEQ  CHANGEDOSINIT ; good...no parms
1F20 D0FE    1670 OOPS   BNE  OOPS      ; otherwise, tough!

```

THE ACTUAL RESET TRAP

```

1F22          1680      .PAGE "THE ACTUAL RESET TRAP"
           1690 ;
           1700 ; On reset, DOS normally gets
           1710 ; called to reinitialize itself...
           1720 ; we use this to our advantage by
           1730 ; reinit'ing both DOS and BASIC
           1740 ;
           1750 RESET
1F22 200000  1760      JSR  0          ; second two bytes get replaced
           1770 ;                      by the address of real DOSINIT
1F25 A2FF    1780      LDX  #$FF
1F27 9A      1790      TXS          ; BASIC likes it this way
1F28 8611    1800      STX  BRKFLAG   ; ensure no BREAK key pending
1F2A 2041BD  1810      JSR  CLOSEALL ; so everybody agrees
1F2D 2072BD  1820      JSR  SETDZ
1F30 A592    1830      LDA  MEOL
1F32 F003    1840      BEQ  RST2
1F34 2099BD  1850      JSR  FIXEOL
           1860 RST2
1F37 2051DA  1870      JSR  INITBUF
           1880 ;
1F3A 20121F  1890      JSR  MOVEMEMLO ; to protect this code
           1900 ;
           1910 ; NOW we fool BASIC into thinking
           1920 ; an error occurred
           1930 ;
           1940 ;
1F3D A9FF    1950      LDA  #255      ; (or your choice of errors)
1F3F 85B9    1960      STA  ERRNUM
1F41 4C40B9  1970      JMP  ERROR     ; do it
           1980 ;
           1990 ; THE FOLLOWING EQUATE IS USED TO SET

```

```

                2000 ; RESETTOP on a page boundary
                2010 ;
2000            2020 RESETTOP = *+$FF&$FF00
                2030 ; SET UP LOAD AND GO
                2040 ;
1F44            2050      *= $2E0
02E0 001F      2060      .WORD CHANGEDOSINIT
02E2            2070      .END

```

THE ACTUAL RESET TRAP

=02E7 MEMLO	=00FF LOW	=0100 HIGH	=BD72 SETDZ
=0092 MEOL	=BD99 FIXEOL	=00B9 ERRNUM	=B940 ERROR
=00BD TRAPFLAG	=DA51 INITBUF	=0011 BRKFLAG	=BD41 CLOSEALL
=000C DOSINIT	1F00 CHANGEDOSINIT	1F22 RESET	1F12 MOVEMEMLO
=2000 RESETTOP	1F1D FROMBASIC	1F20 OOPS	1F37 RST2

Program 2.

```

10 AUDCTL=53768:DEL=120
20 AUDF1=53760:AUDC1=53761
30 SOUND 1,10,10,15:SOUND 3,10,10,
  15
40 POKE AUDC1,0:POKE AUDC1+4,0
50 POKE AUDCTL,DEL
60 FOR J=10 TO 15:POKE AUDF1+2,J:
  POKE AUDF1+6,20-J
70 FOR I=0 TO 255:POKE AUDF1,I:POKE
  AUDF1+4,255-I:NEXT I
80 NEXT J
  ...VERY SMOOTH GLIDES...

```

Program 3.

```

10 AUDCTL=53768:DEL=120
12 OSC=1789790/2
20 AUDF1=53760:AUDC1=53761
30 SOUND 1,10,10,0
40 POKE AUDC1,0:POKE AUDC1+4,0
50 POKE AUDCTL,DEL
60 P2=2^(1/12)
70 NTE=16:REM C IN THE REAL BASS
80 FOR I=1 TO 109
90 FREQ=INT(OSC/NTE-7+0.5):F0=INT
  (FREQ/256)
92 F1=FREQ-256*F0
100 POKE AUDF1,F1:POKE AUDF1+2,F0
102 POKE AUDC1+2,175
103 PRINT "NOW PLAYING ";INT(NTE+
  0.5);" HZ"
105 FOR J=1 TO 100:NEXT J
110 NTE=NTE*P2
120 NEXT I
130 GOTO 70
  ...9 OCTAVE CHROMATIC SCALE...

```

approximately two octaves. Unfortunately, you do not get to have some channels high and others low. Example:

```

SOUND 0,60,10,12 : SOUND 1,45,10,12 : POKE
53768,1 : FOR I=1 TO 500 : NEXT I

```

Again, investigate *De Re* for even more details on some of the more complex aspects of the sound system. (Want to hear your Atari "MOO" like a cow?)

A Final Caution

I have had a couple of people write or call me complaining that, when they tried my assembly language routines, they didn't work. Honest, they *do* work. The listings published in the magazine are the ones I actually used. Sometimes the problem simply resolves to a typo on the part of the user. But sometimes it turns out to be an address conflict.

I do most of my work with OS/A+ and BASIC A+ (naturally. But I use Atari BASIC to check out programs for these pages), and I usually use memory addresses which are convenient to me. Since I get tired of putting everything in page six, I sometimes use \$1F00 or some such as an origin. You *can* use these addresses in your system with the Atari Assembler/Editor if you change MEMLO to, say, \$3000 (my usual choice, and achievable with the LOMEM command of EASMD). However, it may be more convenient to use SIZE to look at where your source, etc. is and then change the origin to reflect your memory configuration.

Of course, you can always assemble into the dreaded page six or assemble directly to disk (or cassette). But, in any case, don't use an origin ("*=xxxx") which conflicts with SIZE in your system. I purposely give you source listings so that you can see how things work; take the time to type them in and it will probably prove easier in the long run.

Insight: Atari

Bill Wilkinson
Optimized Systems Software
Cupertino, CA

The major program for this month is perhaps the most exciting one to appear in this column to date. We will take advantage of Atari's modular software construction to define a set of *soft keys*, a concept that is marketed for real \$\$\$ on some machines. The most obvious use of soft keys is in writing BASIC programs. Even with the abbreviations allowed by BASIC, wouldn't it be convenient to be able to use a single keystroke to get a disk directory listing? And, of course, when programming in assembly language there are certain character combinations that are repeated often enough to justify the use of soft keys (e.g., "Y" or ".BYTE").

The techniques presented in the soft key program include how to "steal" the system's default I/O devices and adapt them for your own purposes. It might be worth your while to re-read my column on adding the "M:" driver, (**COMPUTE!**, January, 1982, #20, pg. 120) since I will be assuming your knowledge of some of the points made there.

For the BASIC user, the soft keys can be made truly "soft" — even to the point of allowing a running BASIC program to change the definition of what a soft key means. And, of course, there will be the usual set of tidbits for those who don't feel up to tackling the soft keys project.

An Announcement

As most of you probably know, magazine articles and columns are written months before they actually appear. As I write this in mid-February, my company (Optimized Systems Software, Inc.) and **COMPUTE!** are frantically engaged in getting a new book ready for publication. *Inside Atari DOS* will presumably have made its appearance by the time you read this. Now, for the first time, Atari users will have access to the listing of the File Manager System of Atari DOS 2.0S, the current version of

Atari DOS. Besides the listings, there is a complete description of each major subroutine, complete with entry and exit parameters and error conditions.

The book is not complete in and of itself: you would still need Atari's listings of the OS ROMs and DUP to have access to all of the DOS secrets. But this book will tie together many loose ends.

Let me leave you with one caveat (don't I always?): the book assumes that the reader has at least a working knowledge of 6502 assembly language. The book is of most value if you would like to see how such a complex organism as a DOS is built.

The terminology "soft" keys refers to keyboard keys that may change "meaning" as desired by the user. The Atari keyboard keys which we will make "soft" include the characters "control-A" through "control-Z" (that is, what are normally graphic characters produced by holding down the control key while hitting one of the letter keys).

The keys will be made soft in a very flexible fashion: each of the 26 keystrokes may be defined in such a way that entering one of them will "fool" the Atari OS (and hence BASIC, etc.) into thinking that a sequence of one or more ordinary keys have been depressed. The phrase "one or more" is literal: there is no effective limit on the number of characters a soft key may represent.

As this program is written, there are some limitations. Only characters with an ATASCII value of 1 to 127 decimal (\$01 to \$7F hex) may be placed in the string. The CR (RETURN or End-Of-Line) character is thus not permitted (since its value is \$9B hex, 155 decimal). Since lack of CR seems to me to be a major flaw, each zero (\$00) byte in the string is converted so that OS sees a CR instead.

The reason that only the values from 0 to 127 are acceptable is that a byte with its most significant bit (MSB, \$80 or 128) turned on is our signal that this byte is to be the last character in the soft key string. Obviously, you can rewrite this part, if you desire, so that some other means is used to designate the end of string (a preceding length byte or trailing zero byte are obvious alternatives). However, the method chosen is simple and seems adequate for most purposes.

For the BASIC user, the soft keys can be made truly 'soft'...

One more note before we get into implementation details: since there are times when you might really want the graphics character "hidden" by a soft key, I have designed an "escape" sequence. Zeroing "control-comma" (normally the graphics escape character) signals to the soft key routine that the next character is *not* to be translated. Thus, even the heart may be generated by pressing control-comma twice.

The Nifty-Gritty

Program 1 shows the complete source of Easykey, the program which implements all the features mentioned above. The program is composed of five primary parts.

The first part, with line numbers in the 2000-2999 range, is used to hook the routine into Atari's DOS. First, we search the Handler Table looking for the E: device. When we find it, we hold onto its address and put the address of our replacement driver in the table instead. Recall that the address in HATABS must be the address of the Handler Routine Table. We copy the current table (presumably the Atari default table, from ROM) to our NEWETBL (new E. table) and replace the entry point for the get-a-single-character routine with the address of our new routine (NEWGETCH) (see one (always required, see commentary on the A: driver in my January column). We then change ROMEM so BASIC won't wipe us out and exit.

The second part of this process is the new get-a-single-character routine for the screen editor (E:) device. Most of this code is copied directly from the DOS ROMs, the only exceptions being the branches to locations in ROM and the call to the keyboard

get-a-character routine (KGETCH).

The third part, NEWKGETCH (NEW Keyboard GET single Character routine), is the heart of this whole process. Here is where individual keystrokes are actually interpreted from their hardware codes and characteristics to a more palatable ATASCII code. But here, also, is where the system is vulnerable to our machinations. Since nothing "downstream" of the keyboard handler (e.g., the rest of the E: driver, CIO, BASIC, etc.) knows what happened at the physical keyboard level, the calling routines will believe the keyboard handler no matter what it tells them.

Actually, NEWKGETCH is fairly simple. First it checks to see if it is already processing a soft key. If so, it simply hands the caller the next key of the soft key's string. If not, it goes and gets a real key from the keyboard. If that key is a heart, it simply gets the next real keyboard key and passes that back to the caller (our "escape" clause). Otherwise, if the keyboard key was not control-A through Control-Z, then the key is returned to the caller unchanged.

If the keyboard key was one of the definable soft keys, its value is used to index into a table of soft key string pointers. Here one last validity check is made: if the string pointer is zero, the keyboard key is returned to the caller and no soft key string handling occurs. If a string pointer is encountered, its address is placed in KPTR and used to access all the characters in the string.

Note that zero bytes are translated to \$9B (CR) characters and that any character with its most significant bit on terminates the string (by zeroing the high order byte of KPTR).

The fourth part of this routine is simply the above-mentioned table of soft key string pointers. Note that we take advantage of the fact that the assembler places zeroes into .WORDS (or .BYTES) which are undefined.

The last part, of course, consists of the actual strings. Note the flexibility here. Control-D (label SD), for example, includes the modified CR character (\$00) as also its last character by simply turning on the MSBit, producing a code of \$80.

Soft Keys: Using Them From BASIC

To conserve space and to show the flexibility of the soft key system, I included strings for only three keys: control-D, which causes a disk directory listing, control-S for "SETCOLOR," and control-P for "PRINT#." The simplest way to add more soft key strings is to put them into the source given (with labels "SA" through "SZ," as appropriate) and assemble the whole thing at an address appropriate for your system.

But you can add or change soft key strings

dynamically from BASIC via POKes, etc. Note that Easykey, as given here, reserves over 450 bytes for soft strings. Note also the addresses of the labels "STABLE" and "STRINGS." If you assemble your own copy of Easykey, be sure and note the addresses of these two labels and convert them to decimal if you intend to use dynamic soft keys.

...if you control these seven pointers, you control BASIC...

Program 2, the BASIC program, is a sample which will allow you to redefine all 26 soft keys to any string you like and then save the resulting definitions in a disk file for later use. Study the technique, and you should be able to produce any kind of soft keys you might want. The program fragment (Program 3) will allow the reloading of predefined sets produced by the previous program.

Inside Atari BASIC: Part 4

This month we will feature a short discussion of the various "tables" used by Atari BASIC and how to find them. Some of this material is well covered by some of the articles in *COMPUTE!'s First Book of Atari*, so if you are too impatient to wait for next month you can run out and buy the book. Next month we will begin to use the information we discover this month to "fool" BASIC into letting us do things it was never designed for. We begin:

When an Atari BASIC program is SAVED to disk or cassette, there are only 14 bytes of zero page written out along with the main tables and program. These 14 bytes consist of seven two-byte pointers (in the traditional low-byte, high-byte form) which tell BASIC where everything is in the particular program being SAVED (or later being LOADED). All the other important zero page locations (and there are over 50 of them) are regenerated and/or recalculated by BASIC anytime you type NEW or SAVE (or, for some locations, RUN, GOTO, etc.). The implication is that, if you control these seven pointers, you control BASIC so let's examine their names and functions. Table 1 gives a summary thereof.

The first thing you may note about this table is that some of the locations (indicated by asterisks)

have duplicate labels. If you examine the mnemonic meanings, you will probably see why: the pointer can mean different things in different contexts. For example, the space pointed to by location \$80 (decimal 128) is used for different purposes, depending on whether BASIC is currently working on entering a new line (it uses OUTBUFF) or executing an expression within a program (when it uses ARGOPS).

You might also note that I provided a list of more than seven pointers. The locations \$8E and \$90 are not SAVED and reLOADED because they are always dependent on the current state of the program (i.e., whether it is RUNNING, whether it has executed a DIM statement, etc.). They are included here for completeness: aside from the zero page locations (and the \$600 page locations with BASIC A+), these pointers completely define BASIC's usage of the Atari computer's memory space. So now let's go into detail about what each of these pointers is used for.

Table 1: BASIC's Critical Zero-Page Pointers

Location		Mnemonic	Which means:
Hex	Decimal	Label:	
80	128	LOMEN	pointer to LOW MEMory limit
80	128 *	ARGOPS	ARGument/OPERator Stack
80	128 *	OUTBUFF	syntax OUTput BUFFer
82	130	VNTP	Variable Name Table Pointer
84	132	VNTD	Variable Name Table Dummy end
86	134	VVTP	Variable Value Table Pointer
88	136 *	STMTAB	STateMent TABLE
8A	138	STMCUR	CURrent STateMent pointer
8C	140	STARP	STring/ARray Pointer
--	---		
8E	142	ENDSTAR	ENDString/ARray space
8E	142 *	RUNSTK	RUNtime STAcK pointer
90	144	TOPRSTK	TOP of Runtime STAcK space
90	144 *	MEMTOP	pointer MEMory TOP limit

We already noted that ARGOPS is used in expression evaluation. That is, whenever BASIC sees any kind of expression to be evaluated [e.g., $3*A + B$ or $SIN(30)$ or $2^{(LOG(4/EXP(Y*Z^3)))-1/(Z^{2.5} + ATN(Z))}$ or even 1.25], it must put intermediate results and/or operators on a "stack." ARGOPS points to a 256 byte area reserved for both the argument stack and the operator stack. (What actually happens in Execute Expression is extremely complex and far beyond the scope of this article.) Since expression evaluation and program entry cannot occur at the same time, OUTBUFF shares this same 256 byte space. When a program line is entered, BASIC checks it for syntax and converts it to internal tokens, placing these tokens temporarily into this 256 byte buffer (before moving them into the appropriate place in the

program, depending on the line number). Again, this process is complex, but the results have been documented here in prior columns and in such places as *De Re Atari* and *COMPUTE!'s First Book of Atari*.

VNTP and VNTD point to the Variable Name Table. In Atari BASIC and BASIC A+, only the first occurrence of a name causes an entry to be added to this table. Within the tokenized program, the name(s) are replaced by a "variable number" which refers to the name's position within the name table. The names are simply placed in this table one after the other, with no intervening bytes, and the end of a name is signaled by turning on the significant bit (\$80, 128 decimal) of its last character. Note that the dollar sign on the end of a string name and the left parenthesis on the end of an array name are included in this table.

VVTP and ENDDVT define the limits of the Variable Value Table. Aside from the actual tokenized BASIC program, this is probably the most interesting of the tables. Each variable occupies eight bytes in this table, so the variable number token need only be shifted left three times to index to the proper location herein. In Part 5 of this series, we will delve into this table in depth, finding many ways to fool BASIC, but there is no room in this issue for more on the subject.

STMTAB defines the beginning of the tokenized program; and, since there is no proper label to refer to, we may consider that STARP defines the end of same. Again, I refer to previous parts of this series for details on the structure of tokenized lines. STMCUR is interesting because it normally points to the actual line currently being executed. This would be one way of implementing special "statements" in Atari BASIC; a USR call would cause the subroutine to use STMCUR to examine the rest of the line for variables, etc. But my comments on ease of use, etc., from last month still apply: I don't think this is really practical.

STARP is the last of the seven pointers that are SAVED and LOADED. Actually, it is included only to point to the end of the program area. The string/array space is *not* SAVED or LOADED (but see Al Baker's article on "Atari Tape Data Files" in *COMPUTE!'s First Book of Atari* for some tricky techniques which I may expand on in future columns). STARP and ENDSTAR define the limits of the string/array space. Atari BASIC is different from Microsoft-style BASICs in that arrays and strings are allocated from this space in the order they are DIMensioned and are not moved around relative to each other after that. Thus, if you code "DIM A\$(100),B(3,3)" then you can use ADR(A\$+100 as the address of B(0,0).

Finally, there is the run-time stack, defined by

RUNSTK and TOPRSTK. When a GOSUB or FOR (or WHILE in BASIC A+) is encountered, the current "address" (consisting of the line number and statement offset within the line) must be "pushed" onto a stack to wait for the corresponding RETURN or NEXT (or ENDWHILE) to "pop" it off, so that the loop or mainline routine may continue where it left off. This stack thus expands and contracts as necessary while a program is running. Again, full details of how the stack is accessed can't be discussed here. In any case, the mechanism is relatively simple for GOSUB and WHILE; only FOR...NEXT presents some interesting problems.

Before we leave this topic for this month, we should note that when a program is SAVED all seven pointers are "relativized" to zero. That is, each pointer has the value of LOMEM (which is also the first pointer) subtracted from it. Then when the program is LOADED, the current value of LOMEM is added to each pointer, thus allowing self-relocating BASIC programs. A side effect of this process is that the first pointer is thus always zero (actually two bytes of zero), and BASIC uses this fact as a self-check when LOADING: it assumes that any file which does *not* start with two zero bytes cannot be a BASIC SAVED program.

Tidbit #4: Structured Programs

An often desirable construct within properly structured programs is this one:

```
1. IF <expression> THEN <procedure-1>
   ELSE <procedure-2>
```

Since BASIC doesn't support procedures, we will modify this to the more familiar-looking form:

```
2. IF <expression> THEN GOSUB <line-1>
   ELSE GOSUB <line-2> or, using BASIC A+,
3. IF <expression> : GOSUB <line-1> ELSE :
   GOSUB <line-2> : ENDIF
```

But still, the Atari BASIC programmer cannot use either of these forms. Take heart! There is a solution which is a logical replacement for 2., above:

```
4. ON <logical-expression> + 1 GOSUB <line-1>, <line-2>
```

Note that there is a subtle difference: where the IF allowed "expression," we now require "logical-expression." The reason is fairly obvious if you recall that a logical expression in Atari BASIC (e.g., A<B or B>=0 or A\$<>B\$) always evaluates to a one (true) or zero (false). By adding one (the "+ 1" in 4.) to a logical expression's value, we have a value of either one or two, something which ON...GOSUB is quite happy with since it GOSUBs to line-1 if the value is one and line-2 if the value is two.

If you really do have an "expression" to replace (e.g., IF A THEN...), simply change it into a logical expression by comparing it to zero, thus:

```
IF A THEN ...
becomes
IF (A<>0) THEN ...
which becomes
ON (A<>0) + 1 THEN ...
```

...we are going to let Atari's CLEAR-SCREEN character do all the work for us.

P.S.: If you want some structuring, but not too much, notice that the GOSUBs in 2. and 4. may be changed to GOTOs with similar effects.

Tidbit #2: A Bug in DOS 2.0S

DOS 2.0S and OS/A+ have an improvement which allows much faster disk reads and writes. When DOS detects that a large data transfer is about to take place, it drops into what is called *Burst I/O Mode*. However, when a file is opened for update (OPEN #1,12,...), burst I/O should not take place. DOS handles update writes correctly, but will often blow it on update reads. The following two, one-byte patches may be made and then DOS should be re-written to the disk (with INIT under OS/A+, with menu option "H" under Atari DOS 2.0S). Caution: do *not* apply these patches to any other versions of DOS!

from BASIC:	from DEBUG:
POKE 2596,144	C A24<90
	C AD5<IF
POKE 2773,31	('C' is the Change command in BUG.)

Tidbit #3: Clearing Memory (Revisited)

My thanks to Jerry White for permission to share his ideas on this with you. This concept is actually the result of a series of coincidences. Coincidence #1: a zero byte in screen memory is displayed as a space on the screen (not true on most machines, where \$20—decimal 32—is the space character). Coincidence #2: the Atari CLEAR-SCREEN character (SHIFT-CLEAR or CTRL-CLEAR) is not subjected to most of the cursor range checks that other characters must go through. Coincidence #3: the code to clear the screen doesn't just clear

one line 24 times (as does, for example, the Apple II's code); instead it simply starts at what it thinks is the lowest address being displayed and continues to the top of memory.

By now, it should be obvious that we are going to let the Atari's CLEAR-SCREEN character do all the work for us. The only thing we must do is fool it into believing that the "screen" is where we want it and is the size we want it.

CLEAR-SCREEN starts clearing at the location pointed to by \$58 (88 decimal) and continues until one-byte short of the page pointed to by \$6A (106 decimal). That is, it always stops clearing at location \$xxFF, where xx is one less than the contents of \$6A. So our memory clear program fragment looks something like this:

```
LOWADDR = ????: REM the lowest address to clear
HIADDR = ????: REM the highest address to clear
!! must end on xxFF boundary !!
* SVLOW1 = PEEK(88) : SVLOW2 = PEEK(89)
SVHI = PEEK(106)
POKE 106,INT( (HIADDR + 1) / 256 )
TEMP = INT(LOWADDR/256) : POKE 89,TEMP
POKE 88,LOWADDR-256*TEMP
PRINT CHR$(125); : REM this does the actual clear
POKE 106,SVHI
* POKE 88,SVLOW1 : POKE 89,SVLOW2
```

Some *cautions* are in order (as usual): 1) The screen editor thinks that it really has cleared the screen and homed the cursor. For safety's sake, it is probably best to follow that code fragment with either a GRAPHICS statement or a real screen clear. 2) Since you can only specify the high (ending) address to the nearest page boundary, you have to be careful you aren't wiping something else out.

For once, though, *caution* number (1) has a good side effect. If you follow the program fragment given above with a GRAPHICS statement, then locations 88 and 89 are going to get recalculated anyway! So the lines marked with asterisks may be omitted in such cases.

P.S.: If you have BASIC A+, there is a much easier method, related to the way strings may be cleared. Given that you know LOWADDR and HIADDR, as in the fragment given above, you may clear the area via the following:

```
poke lowaddr,0 : move lowaddr,lowaddr + 1,
hiaddr-lowaddr (And, wouldn't you know it, another
caution: the system gets very unhappy if
hiaddr = lowaddr.)
```

Next month, we will teach you how to have your Atari take over the entire Bell telephone system. All you need is 37 billion dollars, 25000 miles of #0000 gauge copper wire, and a toothpick. *Caution*: if you try this you musssssssssssssstttttt tabbbbbbbbbbbbeeeeee suuuuuurrrrrr.....asdf aresetasyghxvnbær6q23uqerngt1357 etaoïn shrdlu


```

0090 .PAGE " EQUATES"
1000 ;
1010 ;
1020 ; EASYKEY --
1030 ; A program to ease repetitive typing
1040 ; when using Atari BASIC, etc.
1050 ;
1060 ; Written by Bill Wilkinson
1070 ; Optimized Systems Software
1080 ;
1090 ; for the May, 1982, issue of COMPUTE!
1100 ;
1110 ;
1120 ;
1130 ; Equates to subroutines, etc., located
1140 ; in the Atari OS ROMs
1150 ;
1160 ;
1170 ; CAUTION: these equates are for the
1180 ; revision 'A' ROMs.
1190 ;
1200 SWAP = $FCB3
1210 ERANGE = $FA88
1220 BUFCNT = $6B
1230 ROWCRS = $54
1240 COLCRS = $55
1250 BUFSR = $6C
1260 KGETCH = $F6E2
1270 DSTAT = $4C
1280 ATACHR = $2FB
1290 CR = $9B
1300 EGTC2 = $F66E
1310 EGTC3 = $F67C
1320 LOGCOL = $63
1330 BELL = $F90A
1340 DOSS = $F6AD
1350 RETURL = $F634
1360 ;
1370 ;
1380 HATABS = $31A
1390 SYSTEMLOMEM = $2E7
1400 .PAGE
1500 ; EQUATES UNIQUE TO THIS ROUTINE
1510 ;
1520 ;
1530 QUOTE = $22 ; The " character
1540 NUL = 0 ; A nul...which becomes a CR
1550 ;
1560 ZTEMP1 = $E6 ; shared with fltg pt routines
1570 ;
1580 LOW = $FF
1590 HIGH = $100
1600 KQUIT = $80 ; MSBit says quit
1610 ;
1620 ;
1630 ;
1640 ;
1650 ;
1660 ;
1670 ;
1680 ;
1690 ORIGIN = $1F00
1700 NEWLOMEM = ORIGIN+$300
1710 * = ORIGIN
1720 .PAGE " Hooking the driver into OS"
2000 ;
2010 ;
2020 ;
2030 ; The 'HOOKUP' routine --
2040 ;
2050 ; this portion of the code simply hooks

```

```

2060 ; our replacement driver into the E:
2070 ; handler vector table.
2080 ;
2090 ;
2100 ;
2110 ;
2120 HOOKUP LDX #0 ; start at beginning
2130 LOOKLOOP
2140 LDA HATABS,X
2150 CMP #'E ; looking for the E: driver
2160 BEQ EFND ; got it
2170 INX
2180 INX
2190 INX ; to next name
2200 BNE LOOKLOOP ; keep looking
2210 BEQ HOOKUP ; actually, this is fatal error
2220 ;
2230 ;
2240 ; found the E driver
2250 EFND
2260 LDA HATABS+1,X ; get LSB of addr
2270 STA ZTEMP1
2280 LDA HATABS+2,X ; and MSB
2290 STA ZTEMP1+1
2300 LDA #NEWETBL&LOW
2310 STA HATABS+1,X ; replace with our table
2320 LDA #NEWETBL/HIGH
2330 STA HATABS+2,X
2340 ;
2350 LDY #15 ; all bytes of table
2360 EMVLP
2370 LDA (ZTEMP1),Y ; get a byte of old tbl
2380 STA NEWETBL,Y ; to our table
2390 DEY
2400 BPL EMVLP ; and do more
2410 ;
2420 ; E: handler vector table is moved
2430 ;
2440 LDA #NEWGETCH-&LOW
2450 STA NEWETBLGC ; change the get character ptr
2460 LDA #NEWGETCH-1/HIGH
2470 STA NEWETBLGC+1
2480 ;
2490 ; tables, etc. are corrected
2500 ;
2510 ; move lomem
2520 ;
2530 ;
2540 LDA #NEWLOMEM&LOW
2550 STA SYSTEMLOMEM
2560 LDA #NEWLOMEM/HIGH
2570 STA SYSTEMLOMEM+1
2580 RTS
2590 .PAGE " The replacement E: driver"
3000 ;
3010 ;
3020 ;
3030 ; Begin the actual E: replacement routine
3040 ;
3050 ; This routine replaces E's get-a-character
3060 ; entry point.
3070 ;
3080 ;
3090 ;
3100 ;
3110 NEWGETCH
3120 JSR SWAP ; << lines with comments starting
3130 JSR ERANGE ; << with "<<" are copied from
3140 LDA BUFCNT ; << without change from ROM
3150 ;
3160 BNE GOGETC3 ; was just EGTC3
3170 ;
3180 LDA ROWCRS ; <<
3190 ;

```

.WORD SS,ST,SU,SV,SW,SX,SY,SZ

```

202A 0000 REM FIRST, CLEAR OUT OLD SOFTKEY DEFINITIONS
202C 0000 REM
202E 0000 REM
2030 0000 REM
2032 0000 REM
2034 0000 REM
1240 REM
1260 REM
1280 REM
1300 REM
1320 REM
1340 REM
1360 REM
1380 REM
1400 REM
1420 REM
1440 REM
1460 REM
1480 REM
1500 REM
1520 REM
1540 REM
1560 REM
1580 REM
1600 REM
1620 REM
1640 REM
1660 REM
1680 REM
1700 REM
1720 REM
1740 REM
1760 REM
1780 REM
1800 REM
1820 REM
1840 REM
1860 REM
1880 REM
1900 REM
1920 REM
1940 REM
1960 REM
1980 REM
2000 REM
2020 REM
2040 REM
2060 REM
2080 REM
2100 REM
2120 REM
2140 REM
2160 REM
2180 REM
2200 REM
2220 REM
2240 REM
2260 REM
2280 REM
2300 REM
2320 REM
2340 REM
2360 REM
2380 REM
2400 REM
2420 REM
2440 REM
2460 REM
2480 REM
2500 REM
2520 REM
2540 REM
2560 REM
2580 REM
2600 REM
2620 REM
2640 REM
2660 REM
2680 REM
2700 REM
2720 REM
2740 REM
2760 REM
2780 REM
2800 REM
2820 REM
2840 REM
2860 REM
2880 REM
2900 REM
2920 REM
2940 REM
2960 REM
2980 REM
3000 REM
3020 REM
3040 REM
3060 REM
3080 REM
3100 REM
3120 REM
3140 REM
3160 REM
3180 REM
3200 REM
3220 REM
3240 REM
3260 REM
3280 REM
3300 REM
3320 REM
3340 REM
3360 REM
3380 REM
3400 REM
3420 REM
3440 REM
3460 REM
3480 REM
3500 REM
3520 REM
3540 REM
3560 REM
3580 REM
3600 REM
3620 REM
3640 REM
3660 REM
3680 REM
3700 REM
3720 REM
3740 REM
3760 REM
3780 REM
3800 REM
3820 REM
3840 REM
3860 REM
3880 REM
3900 REM
3920 REM
3940 REM
3960 REM
3980 REM
4000 REM
4020 REM
4040 REM
4060 REM
4080 REM
4100 REM
4120 REM
4140 REM
4160 REM
4180 REM
4200 REM
4220 REM
4240 REM
4260 REM
4280 REM
4300 REM
4320 REM
4340 REM
4360 REM
4380 REM
4400 REM
4420 REM
4440 REM
4460 REM
4480 REM
4500 REM
4520 REM
4540 REM
4560 REM
4580 REM
4600 REM
4620 REM
4640 REM
4660 REM
4680 REM
4700 REM
4720 REM
4740 REM
4760 REM
4780 REM
4800 REM
4820 REM
4840 REM
4860 REM
4880 REM
4900 REM
4920 REM
4940 REM
4960 REM
4980 REM
5000 REM
5020 REM
5040 REM
5060 REM
5080 REM
5100 REM
5120 REM
5140 REM
5160 REM
5180 REM
5200 REM
5220 REM
5240 REM
5260 REM
5280 REM
5300 REM
5320 REM
5340 REM
5360 REM
5380 REM
5400 REM
5420 REM
5440 REM
5460 REM
5480 REM
5500 REM
5520 REM
5540 REM
5560 REM
5580 REM
5600 REM
5620 REM
5640 REM
5660 REM
5680 REM
5700 REM
5720 REM
5740 REM
5760 REM
5780 REM
5800 REM
5820 REM
5840 REM
5860 REM
5880 REM
5900 REM
5920 REM
5940 REM
5960 REM
5980 REM
6000 REM
6020 REM
6040 REM
6060 REM
6080 REM
6100 REM
6120 REM
6140 REM
6160 REM
6180 REM
6200 REM
6220 REM
6240 REM
6260 REM
6280 REM
6300 REM
6320 REM
6340 REM
6360 REM
6380 REM
6400 REM
6420 REM
6440 REM
6460 REM
6480 REM
6500 REM
6520 REM
6540 REM
6560 REM
6580 REM
6600 REM
6620 REM
6640 REM
6660 REM
6680 REM
6700 REM
6720 REM
6740 REM
6760 REM
6780 REM
6800 REM
6820 REM
6840 REM
6860 REM
6880 REM
6900 REM
6920 REM
6940 REM
6960 REM
6980 REM
7000 REM
7020 REM
7040 REM
7060 REM
7080 REM
7100 REM
7120 REM
7140 REM
7160 REM
7180 REM
7200 REM
7220 REM
7240 REM
7260 REM
7280 REM
7300 REM
7320 REM
7340 REM
7360 REM
7380 REM
7400 REM
7420 REM
7440 REM
7460 REM
7480 REM
7500 REM
7520 REM
7540 REM
7560 REM
7580 REM
7600 REM
7620 REM
7640 REM
7660 REM
7680 REM
7700 REM
7720 REM
7740 REM
7760 REM
7780 REM
7800 REM
7820 REM
7840 REM
7860 REM
7880 REM
7900 REM
7920 REM
7940 REM
7960 REM
7980 REM
8000 REM
8020 REM
8040 REM
8060 REM
8080 REM
8100 REM
8120 REM
8140 REM
8160 REM
8180 REM
8200 REM
8220 REM
8240 REM
8260 REM
8280 REM
8300 REM
8320 REM
8340 REM
8360 REM
8380 REM
8400 REM
8420 REM
8440 REM
8460 REM
8480 REM
8500 REM
8520 REM
8540 REM
8560 REM
8580 REM
8600 REM
8620 REM
8640 REM
8660 REM
8680 REM
8700 REM
8720 REM
8740 REM
8760 REM
8780 REM
8800 REM
8820 REM
8840 REM
8860 REM
8880 REM
8900 REM
8920 REM
8940 REM
8960 REM
8980 REM
9000 REM
9020 REM
9040 REM
9060 REM
9080 REM
9100 REM
9120 REM
9140 REM
9160 REM
9180 REM
9200 REM
9220 REM
9240 REM
9260 REM
9280 REM
9300 REM
9320 REM
9340 REM
9360 REM
9380 REM
9400 REM
9420 REM
9440 REM
9460 REM
9480 REM
9500 REM
9520 REM
9540 REM
9560 REM
9580 REM
9600 REM
9620 REM
9640 REM
9660 REM
9680 REM
9700 REM
9720 REM
9740 REM
9760 REM
9780 REM
9800 REM
9820 REM
9840 REM
9860 REM
9880 REM
9900 REM
9920 REM
9940 REM
9960 REM
9980 REM
10000 REM

```

Program 2.

```

1000 REM *****
1010 REM * EASYLOAD --
1020 REM *
1030 REM * A BASIC PROGRAM THAT ALLOWS
1040 REM * THE USER TO MAKE HIS/HER OWN
1050 REM * SET OF "SOFTKEYS" FOR USE
1060 REM * WITH THE "EASYKEY" PROGRAM
1070 REM *
1080 REM *****
1090 REM
1100 REM :BEFORE RUNNING THIS PROGRAM, BE SURE
1110 REM :THAT "EASYKEY" HAS BEEN LOADED AND
1120 REM :RUN. YOU MAY VERIFY THIS BY CHECKING
1130 REM :THE VALUE OF PEEK(744) -- IT SHOULD
1140 REM :MATCH THE NEWLOMEM PAGE VALUE IN
1150 REM :THE LISTING OF EASYKEY
1160 REM :[ PEEK(744) = 34 IF EASYKEY IS
1170 REM :ASSEMBLED AS GIVEN HERE ]
1180 REM
1190 REM == FIRST, SET UP ADDRESSES, ETC. ==
1200 STABLE=8192:REM STABLE = $2000
1210 STRINGS=STABLE*(2*26)
1220 REM
1230 DIM KEYS(130)

```

Program 3.

```

30000 REM *****
30010 REM
30020 REM THIS PROGRAM OR PROGRAM FRAGMENT IS
30030 REM INTENDED TO BE USED TO RELOAD ONE
30040 REM THE SETS OF SOFTKEYS CREATED BY
30050 REM "EASYLOAD".
30060 REM
30070 REM STABLE=8192:REM STABLE=$2000
30080 REM
30090 REM DIM NAMES(20)
30100 REM
30110 REM "What set of softkeys should be"
30120 REM "reloaded from disk (1-999)";:INPUT SET
30130 REM
30140 REM "D:SOFTKEY.";:KEYS(LEN(SET)+1)=STR$(SET)
30150 REM
30160 REM "EASYLOAD".
30170 REM
30180 REM
30190 REM
30200 REM
30210 REM
30220 REM
30230 REM
30240 REM
30250 REM
30260 REM
30270 REM
30280 REM
30290 REM
30300 REM
30310 REM
30320 REM
30330 REM
30340 REM
30350 REM
30360 REM
30370 REM
30380 REM
30390 REM
30400 REM
30410 REM
30420 REM
30430 REM
30440 REM
30450 REM
30460 REM
30470 REM
30480 REM
30490 REM
30500 REM

```

Insight: Atari

Bill Wilkinson
Optimized Systems Software
Cupertino, CA

This month has been a most hectic one. We just finished exhibiting both our new and old products at the seventh West Coast Computer Faire. (The seventh? Is that possible? I remember attending the first!) And, of course, we saw many, many, many new products for Atari Computers there. (Oh, all right, there were some for those other brands, also.) As I have said before, I won't review other companies' software products in this column, but I hope my dear editor won't object if I mention some of the more prominent new hardware products. Presumably, we will be seeing full blown reviews of these products in these pages in the future. And, since **COMPUTE!** was also there, I won't do more than just the mentions.

New Atari Peripherals

There were two companies there with add-on disk drives for the Atari: Percom Data Corporation and MPC Peripherals. It is hoped that both will be delivering double density drives by the time you read this, and the word is that we can expect double-sided, double-density very soon.

32K Byte memory cards were in abundance. And, of course, there was already Axlon's RAM-DISK. And how about a 64K card for the Atari 400? It's available now in Germany. I'm not sure when and/or how it will appear here.

The long-awaited 24-by-80 display (24 lines of 80 characters, instead of Atari's 40 characters) was shown by BIT3 Corporation (who make a similar board for the Apple II).

And Stargate Enterprises (an Atari dealer near Pittsburgh, PA) brought and demonstrated the most innovative prototype: a small, radio-controlled robot. This might not sound exciting until you realize that the controlling end of the radio link was being driven by an Atari.

And wouldst that I could go into the software. Some of the latest arcade games have been, or are being, converted to Atari. And many of the best Apple II games will shortly appear for us, also. The best is yet to come, I believe. My aching pocketbook.

Anyway... as a consequence of all this, I simply didn't have time this month to do a fancy, full-blown program like last month's. Instead, I will just note a couple of the things I've been carrying around on

spare scraps of paper before they get lost. But this won't be a short column; part five of my series on the internals of Atari BASIC is a fairly long and complex article on how variables are used and accessed and more. But first, the tidbits.

Control One Atari Screen

I am constantly amazed at the number of Atari owners (and not necessarily new owners) who are not aware that you can temporarily halt text screen output. They are forever typing LIST (for example) and then trying to hit the BREAK key at exactly the right time. For shame! You didn't read your manuals.

To temporarily pause, simply hit CONTROL-1 (hold down the CTRL key and hit the numeral 1 key). To continue, hit CONTROL-1 again. That's all there is to it.

Now, don't you feel silly? Would it help if I told you that somebody had to tell me, too?

Y Not Do It Later?

There is a minor, but terribly frustrating, bug in the Atari Assembler/Editor cartridge. There is no fix, but it is relatively easy to avoid if one is aware of it. So, if you haven't already been bitten, here is some bug repellent.

The problem has to do with using the Compare-Y immediate instruction (CPY #xxx) when using the cartridge's debugger. One cannot always Step or Trace through such an instruction. Usually, an attempt to do so will cause the instruction to be treated as a BReaK (though I have heard tales of systems crashing).

The sort-of-a-solution is simply to avoid the instruction altogether. If possible, use CPX instead. Or try the following:

WAS:	NOW:
CPY #7	CPY VALUE7
	...
	VALUE7.BYTE 7

This new method eats up two more bytes of memory, but the CPY# should be a fairly rare instruction so this technique won't make a lot of difference.

Using Print Without Using

Every now and then, I see a routine listed and/or used that is supposed to simulate PRINT USING on a BASIC that doesn't have such a capability. (For those of you who don't know what PRINT USING is, suffice to say that it is a very nice tool which allows beautifully formatted numeric output.) Well, I couldn't let these routines go unchallenged, since I had also designed such a routine many years ago. So here is that routine spruced up for Atari BASIC:

```
32000 REM formatted money
32010 TRAP 32020 : DIM QNUM$(15) : TRAP 40000
```



```

32020 IF ABS(QNUM) >= 1E8 THEN QNUM$=
      STR$(QNUM) : RETURN
32030 QNUM$="": IF QNUM<0 THEN
      QNUM=-QNUM : QNUM$="("STR$(INT(QNUM))")"
32040 QNUM$(11-LEN(STR$(INT(QNUM))),10)=
      STR$(INT(QNUM))
32050 QNUM$(11,13)=STR$(100+INT((QNUM-INT
      (QNUM))*100+0.5)): QNUM$(11,11)="." :
      RETURN

```

Alternatively, you might replace the last statement of line 32030 with

```
QNUM$(14,15)="CR"
```

NOTE: to facilitate your counting, I have used an up arrow ("^") where you should type a space.

To use the routine, simply place the number you want formatted into QNUM and GOSUB 32000. The routine returns with the formatted string in QNUM\$. Some things to observe about the routine: it uses no temporary variables, it dimensions its own string (but only once; notice the TRAP), it could be easily translated to any Microsoft BASIC that allowed MID\$ on the left side of the equal sign.

Inside Atari BASIC: Part V

Last month we discussed the seven main memory pointers used by Atari BASIC and BASIC A+, and I promised to make the variable table the main topic for this month. In addition, I said that we would learn how to fool BASIC in useful ways. Many of the techniques I will present this month are *not* my original ideas: I must credit many sources, including *De Re Atari* and *COMPUTE!'s First Book of Atari*. However, the material bears repeating; and perhaps I can give some deeper insight into why and how some of the tricks work.

The Structure Of The Variable Value Table

Please recall from previous articles that the variable value table (VVT) of Atari BASIC is kept distinct from the variable name table. The reason for this is to speed run-time execution. Recall that the tokenized version of a variable is simply the variable's number plus 128 (80 hex), resulting in variable tokens with values from 128 to 255 (\$80 to \$FF). Since each entry in the VVT is eight bytes long, the conversion from token to address within VVT is fairly simple. For those of you who are interested, the following code segment is a simplified version of the actual code as it appears in BASIC:

```

; we enter with the token value
; ($80 through $FF) in A register
;

```

```

LDY #0
STY ZTEMP+1 ;a zero page temporary
ASL A ;token value * 2

```

```

; but ignore the high bit
ASL A ; token value * 4
ROL ZTEMP+1 ; carried into MSB also
ASL A ; token value * 8
ROL ZTEMP+1 ; again, into MSByte
CLC ; (not needed) ... included for clarity
ADC VVTP ; add in LSB of VVT Pointer
STA ZTEMP ; gets LSB of pointer to var
LDA ZTEMP+1
ADC VVTP+1 ; add the two MSBs
STA ZTEMP+1 ; to obtain complete pointer
LDA (ZTEMP),Y ; see text

```

When we exit this routine, ZTEMP has become a zero-page pointer that points to the appropriate eight-byte entry within the variable value table. But just what does it point to? The A-register contains the first byte of that entry. What is that first byte? Read on...

Since each entry in the VVT is eight bytes long (yet may be a simple numeric variable, a string, or an array) obviously the entries must vary in contents. However, the first two bytes always have the same meanings. In particular, the first byte is the "flags" byte, and the second byte is a repeat of the variable number (without the MSBit on). We could probably have dispensed with the repeat of the variable number; but including that byte made the entry size come out to eight bytes (more convenient), and we found several uses for it in the actual implementation.

The "flags" byte is the heart of the whole VVT scheme: until BASIC examines a variable's flag byte, it doesn't even know whether it is working with a string, array, or scalar. But note how neatly we managed to arrive at the end of the routine above with the appropriate flag byte safely in the A-register, where it can easily be checked, compared, or whatever. This, then, is the meaning of the individual bits within the flags byte:

Bit Number	Hex Value	Meaning (if bit is on)
0	\$01	Array or String is DIMensioned
6	\$40	this is an Array
7	\$80	this is a String

Note that there is no special flag that says "this variable is a simple scalar numeric." Instead, the absence of all flags (i.e., a \$00 byte) is used to indicate such variables. Since we have now used the first two bytes of each VVT entry, we now have to figure out what to do with the remaining six bytes. It is no coincidence that Atari floating point numbers consist of six bytes (a one byte exponent and a five byte mantissa): that numeric size was purposely chosen as one that gave a reasonable degree of accuracy as well as reasonable efficiency on the VVT layout. (Yes, I know, seven bytes would have worked well also, especially if we hadn't used the

redundant variable number. Oh well.)

So scalar numeric variables obviously have their value contained directly in the VVT (hence the name, variable *value* table). But what about strings and arrays, which might be any size? The answer is yet another set of pointers, etc. Before proceeding, let us examine the layout of the three kinds of VVT entries, including the already-discussed scalar type:

BYTENUMBER	0	1	2	3	4	5	6	7
SCALARS	00	vnum	(floating point #, 6 bytes)					
STRINGS	80/81	vnum	address	LENGTH	DIM			
ARRAYS	40/41	vnum	address	DIM1+1	DIM2+1			

For strings and arrays, byte zero (the flag byte), varies depending upon whether or not the variable has yet been DIMensioned. (Incidentally, BASIC always resets bit zero of the flag byte and zeros bytes two through seven for all variables whenever you tell it to RUN a program.)

The "address" in bytes two and three of string and array variables is *not* the actual address where the string or array is located. Instead, it is actually an offset (or, if you prefer, relative address) within the string/array space allocated to the program. Recall from last month that location \$8C (140 decimal), names STARP (STring and ARray Pointer), points to the base location of such allocated space. Thus, for example, when BASIC receives a request for "ADR(XX\$)", it simply uses the variable number (for XX\$, which was generated when the program was typed in) to index into VVT (as above), and then retrieves the "address" from the VVT entry and adds it to the current contents of STARP.

For strings, the length and dimension values seem obvious: the DIM value is what you specify with the BASIC DIM statement, and the length is the same as that returned by the LEN function.

For arrays, we need note that DIM1 and DIM2 are as specified by the programmer in the DIM statement [e.g., DIM ARRAY(3,4)]. The reasons they are incremented by one in VVT are twofold: a zero value is used to indicate "dimension not in use" (obviously only effective for DIM2, since flag bit 0 will not be set if neither is in use) also, since the zeroeth element of an array is accessible (whereas the zeroeth character of a string is not), using DIM + 1 makes out-of-range comparisons easier.

And that's it. There really are no other magic tricks or secrets. Once DIMensioned, strings and arrays don't change their offsets (relative addresses) or dimensions. There are no secret flag bits that mean funny things. Turning on the MSBit of the variable number only spells disaster. I really have told all.

Making Use Of What We Know

BASIC is not smart enough to check entries in these tables for validity. It assumes that once you have declared and/or DIMensioned a variable the VVT entry is correct (it must be...BASIC made it so). Thus the implication is that one can go change various values in VVT and BASIC will believe the changes. So let's examine what we can change and what effects (good and bad) such changes will have.

First, as usual, some cautions: BASIC DIMensions variables in the order the programmer specifies. Thus "DIM A\$(100),B(10)" will ensure that the address of array B will be 100 higher than that of string A\$. Neat, sweet, petite. *However*, the order in which variables appear in the VVT (and Variable Name Table) depends entirely upon the order in which the user ENTERED his program. An example:

NEW

```
20 A=0
40 DIM B$(10)
10 DIM C$(10)
30 DIM D(10)
```

LIST

[and BASIC responds with:

```
10 DIM C$(10)
20 A=0
30 DIM D(10)
40 DIM B$(10)
```

Assuming that you typed in the lines above in the order indicated, the variables shown would appear in VVT in alphabetical order (A,B\$,C\$,D). But, if you RUN the program, the DIMensioned variables would use string/array space as follows:

```
C$, 10 bytes, offset 0 from STARP
D(), 66 bytes, offset 10 from STARP
B$, 10 bytes, offset 76 from STARP
```

Though you can figure out this correspondence (especially if you list the variable name table, with a short program in Atari BASIC or with LVAR in BASIC A+), it is probably not what you would most desire. It would be handy if the VVT order and the string/array space order were the same. Solution: (1) Place all your DIMensions first in the program, ahead of all scalar assignments. (2) LIST your program to disk or cassette, NEW, and reENTER — thus insuring that the order you see the variables in your program listing is the same order that they appear in the VVT. From here on in this article I will assume that you have taken these measures, so that variable number zero is also the first variable DIMensioned, etc.

So let's try making our first change. The simplest thing to change is STARP, the master STring/ARray Pointer. A simple program is prob-

ably the easiest way to demonstrate what we can do:

```

100 DIM A$(24*40) : A$(24*40)=CHR$(0)
110 WAIT = 900
120 A$(1,24)="THIS IS ORIGINAL A$ !!! "
130 A$(25)=A$
140 PRINT A$ : GOSUB WAIT
150 SAV140=PEEK(140) : SAV141=PEEK(141)
160 TEMP = PEEK(560)+256*PEEK(561) + 4
170 POKE 140,PEEK(TEMP) : POKE 141,PEEK
    (TEMP+1)
180 PRINT CHR$(125);
190 A$(1,11)="HI there..." : GOSUB WAIT
200 A$(12)=A$ : GOSUB WAIT
210 POKE 140,SAV140 : POKE 141,SAV141
220 PRINT A$
230 END
900 REM WAIT SUBROUTINE
910 POKE 20,0 : POKE 19,0
920 IF NOT PEEK(19) THEN 920
930 RETURN

```

BASIC A+ users might prefer to delete line 160 and change the following lines:

```

150 sav140 = dpeek(140)
170 dpoke 140,dpeek(dpeek(560)+4)
210 dpoke 140,sav140
910 dpoke 19,0

```

"Simple", he said. Who's he kidding!" Honest, it's simpler than it looks. Lines 100 through 140 simply initialize A\$ to an identifiable, printable string and print it. The WAIT routine is simply to give you time to see what's happening. Note that A\$ is DIMensioned to exactly the same size (in bytes) as screen memory. We then save BASIC's STARP value and replace it with the address of the screen (lines 150 through 170). Since A\$ is the first item in string/array space, its offset is zero. Thus pointing STARP to the screen points A\$ to the screen.

We then clear the screen and initialize A\$ again – to a short string. Notice the effect on the screen: capital letters and symbols are jumbled because of the translation done on characters to be displayed. (Recall that Atari has three different internal codes: keyboard code, ATASCII code, and screen code. Normally we are only aware of ATASCII, since the OS ROMs do all the conversions for us.)

At line 200, we proliferate our short string throughout all of A\$ – look at the effect on the screen. Finally, lines 210 through 230, we restore STARP to its original value and print what BASIC believes to be the value of A\$. Surprised?

As interesting as all the above is, it is of at best limited use: moving all of string/array space at once is dangerous. In our example above, if there had been a second string DIMensioned, it would have been reaching above screen memory, into never-never land. Let me know if you can find a real use for the technique.

A better technique would be one which would allow us to adjust the addresses of individual strings (or arrays). While a little more complex, the task is certainly doable. Our first task is to find a variable's location in the VVT. If the variable number is "n", then its VVT address is [VVTP]+8*n (where "[...]" means "the contents of ...").

In BASIC:

```
PEEK(134)+256*PEEK(135)+8*n
```

or BASIC A+ :

```
dpeek(134)+8*n
```

We can then add on the byte offset to the particular element we want and play our fun and games.

Again, a sample program might be the best place to start:

```

100 DIM A$(1025),B$(1025) : A$(1025)=CHR$(0) :
    B$=A$
110 STARP=PEEK(140)+256*PEEK(141)
120 VVTP=PEEK(134)+256*PEEK(135)
130 CHARSET=14*4096 : REM HEX E000
140 VNUM=1 : REM the variable number of B$
150 LET NEWOFFSETB=CHARSET - STARP
160 TEMP1=INT(NEWOFFSETB/256)
170 TEMP2=NEWOFFSETB - 256*TEMP1
180 POKE VVTP+VNUM*8+2,TEMP2 : POKE
    VVTP+VNUM*8+3,TEMP1
190 A$=B$
200 PRINT ADR(B$),CHARSET

```

optionally, in BASIC A+ :

```

100 dim a$(1024),b$(1024) : a$(1024)=chr$(0) : b$=a$
110 starp=dpeek(140)
120 vvtp=dpeek(134)
130 charset=14*4096
140 vnum=1
180 dpoke vvtp+vnum*8+2,charset-starp
190 a$=b$
200 print adr(b$),charset

```

equivalently:

```

100 DIM A$(1024)
110 CHARSET=14*4096
120 FOR I=1 TO 1024
130 A$(I)=CHR$(PEEK(CHARSET+I-1))
140 NEXT I

```

or again, optionally, in BASIC A+ :

```

100 dim a$(1024) : a$(1024)=chr$(0)
110 move 14*4096, adr(a$), 1024

```

The intent of all four of the above program fragments is the same: to move the Atari character set font from ROM (at \$E000) into the string A\$. The third method will probably be the most familiar to most of you. Unfortunately, it is also the slowest. The fourth method, admittedly is clearest in BASIC A+ , though: its line 110 summarizes what we are trying to do in each of the other three.

The first method is of course the one which deserves our attention since it relates to this article. Line 100 simply allocates and initializes our

two strings. We must DIMension these strings one greater than we need because of the bug in Atari BASIC which moves too few bytes when string movements involve moving exact multiples of 256 bytes. Lines 110 and 120 simply get the current values of the two pointers that we need, VVTP and STARP.

Lines 130 and 140 actually simply set up some constants. The Atari character set is always located at \$E000, of course. The VNUM is set to one, in accordance to what we noted above. *Be careful!* The VNUM will *not* necessarily be one if you did not type this program in the order shown! When all else fails, use LIST and reENTER.

We use line 150 to figure out how much B\$ must move (and it will always move "up," since the ROM is always above the RAM) and then calculate its new "offset" within STARP. Of course, it is now actually outside of string/array space, but BASIC doesn't know that. Why should it care?

Unfortunately, lines 160 and 170 are needed in Atari BASIC (and most other BASICs) to manipulate 16-bit numbers into digestible, byte-sized pieces.

Finally, with line 180 we establish B\$ as pointing to the character set memory. Line 190 moves the entire 1025 bytes, with one simple operation, from there to the waiting arms of A\$, in RAM, where it can be manipulated.

With Atari BASIC (and, indeed, with most BASICs), the only other way to get the speed demonstrated here is to write an assembly language subroutine to do the move. Obviously, if you were simply moving the character set once, this is not the way to do it. But if you are interested in manipulating a lot of different memory areas with great speed (player missile graphics? multiple screens?), this works.

A couple of comments: We did not really need to DIMension



GALACTIC CHASE™

The aliens have swept undefeated across the galaxy. You are an enterprising star ship captain—the final defender of space.

As the aliens attack, you launch a deadly barrage of missiles. Flankers swoop down on your position. Maneuvering to avoid the counterattack, you disintegrate their ships with your magnetic repellers. As your skill improves, the attackers increase their speed. And as a last resort, the aliens use their invisible ray to slow the speed of your missile launcher.

GALACTIC CHASE provides Atari owners with the most challenging one or two person game in the galaxy.



Atari 400/800 16k. Written in machine language. Requires joysticks.

Payment: Personal Checks—allow three weeks to clear.

American Express, Visa, & Master Charge—include all numbers on card. Please include phone number with all orders. 24.95 for cassette or 29.95 for disk plus 2.00 shipping. Michigan residents add 4%.

Check the dealer in your local galaxy. Dealer inquiries encouraged.

Galactic Chase © 1981 Stedek Software.

SPECTRUM
COMPUTERS

Dept C.
26618 Southfield
Lathrup Village, MI. 48076
(313) 559-5252

and set up B\$ in our example. After all, as long as we are faking the address, why not fake the DIMension, LENGTH, and flags as well? We could accomplish all that this way:

POKE VVTP+8*VNUM, 65: REM say B\$ is dimensioned (\$41), see above)
POKE VVTP+8*VNUM+4,1: REM 1sb of 1025 (\$0401), the length
POKE VVTP+8*VNUM+5,4: REM msb of ditto
POKE VVTP+8*VNUM+6,1: REM and the DIM is the same as the len
POKE VVTP+8*VNUM+7,4: REM msb of the DIMM

Now we have fooled BASIC into thinking B\$ is set up properly but we haven't actually used any memory for it. P.S.: can you think of any reasons to have two variables pointing to the same memory space? A string and an array pointing the same space? We'll discuss all that next month.

COMPUTE! Subscriber Services

Please Help us serve you better; if you need to contact us for any of the reasons listed below, write to us at:

COMPUTE! Magazine
P.O. Box 5406
Greensboro, NC 27403

or call the Toll Free number listed below.

Change Of Address. Please allow us 6-8 weeks to effect the change; send your current mailing label along with your new address.

Renewal. Should you wish to renew your **COMPUTE!** subscription before we remind you to, send your current mailing label with payment or charge number or call the Toll Free number listed below.

New Subscription. A one year (12 month) US subscription to **COMPUTE!** is \$20.00 (2 years, \$36.00; 3 years, \$54.00. For subscription rates outside the US, see staff page). Send us your name and address or call the Toll Free number listed below.

Delivery Problems. If you receive duplicate issues of **COMPUTE!**, if you experience late delivery or if you have problems with your subscription, please call the toll Free number listed below.

COMPUTE!
800-334-0868
In NC 919-975-9809

QUALITY PRODUCTS FOR ATARI COMPUTERS

SWIFTWARE

FROM SWIFTY SOFTWARE

ORDERED BY NASA

Yes, the National Aeronautics and Space Administration selected these three quality products.

HARDWARE DISK SENTRY™

SAVE
An intelligent digital accessory for your ATARI 810 Disk Drive, lets you selectively write data to both sides of single sided and write protected disks. DISK SENTRY cannot harm your drive or disks. Installs and removes easily; no soldering required. DISK SENTRY's LED signals system status, preventing accidental erasure of data. DISK SENTRY is a convenient push button write-protect override which can pay for itself with your first box of disks. \$39.95

NEW GAMES™ HAUNTED HILL™

NEW!
In this super game you fight bats and ghosts in the dark of a cemetery. This exciting, all machine language game has arcade quality graphics and speed. Requires Joystick. DISK ONLY. \$29.95

TRIVIA TREK™

A multiple choice game of trivia for one or two players or teams. Comes complete with fifty categories of trivia questions and over two thousand multiple choice answers. A program for creating your own trivia questions and answers is also included. Play it for fun, test your knowledge or entertain friends. Parents and teachers can also make up questions. Great at parties. 32K Disk Only. \$29.95

UTILITIES DISKETTE INVENTORY SYSTEM™

Use this system to gain control of your expanding disk/program inventory. Quickly get locations of single or multiple copies of your programs and all your valuable files. An invaluable tool, this system is easy and convenient to use and to update. 24K disk system required. \$24.95 Printer suggested.

SWIFTY UTILITIES

A valuable collection of programming utilities for the ATARI programmer. This DISK ONLY package includes all of Programming Aids I and additional programs designed to make programming time more efficient. Special MENU program runs both saved and listed programs. REM REMOVER eliminates REM statements so programs take less core and run faster. PRINT 825 and PRINTEPS custom print programs prepare condensed, indented and paginated program listings on your ATARI 825 or EPSON MX-80 printer. Listings identify machine code, graphics and inverse video characters. VARIABLE LIST and VARIABLE PRINT programs help you prepare alphabetized annotated list of your program variables. A delete lines utility provides convenience of line deletion while a DOS CALLER gives you convenient access to many DOS utilities while your program is in core. Disklist prepares disk jacket labels. Many of these programs work core-silent with each other and with your program. Disk Drive and minimum of 24K required. \$29.95

PROGRAMMING AIDS PACKAGE I™

Four utility programs to help increase programming efficiency and learn more about your computer. RENUMBER handles references and even variables. Generates Diagnostic Tables for programming error detection. PROGRAM DECODER, DECIMAL to BCD and BCD to DECIMAL programs give you a practical way of studying internal program representation and ATARI number-conversion procedures. Comes with comprehensive user's manual. 16K cassette \$14.95; 24K disk \$19.95

SWIFTY TACH MASTER™

An accurate disk speed diagnostic utility program designed specifically for ATARI 810 Disk Drives. Provides easy-to-read visual indication of the speed of any drive connected to your system. Using the accuracy of machine language, TACH MASTER displays five RPM readings per second with a working tachometer accurate to 1/4 RPM. Allows you to adjust your drive(s) to factory specs easily and at any time in the convenience of your own home. Comes complete with easy to follow user's manual. \$29.95

SWIFTY DATALINK™

High Quality Smart Terminal Communications program. Easy to use Multi-Option, Menu Driven. Full performance uploading/downloading. Works in Duplex or Simplex modes supporting ASCII and ATASCII transmission. Printer Dump, Screen Dump and Disk Search options. Use as remote terminal. Send/receive and store programs and data files. Saves connect time charges with commercial services. Requires 24K RAM, 810 Disk Drive, 850 Interface or equivalent, 830 or other 300 Baud modem. (Printer optional) \$39.95

PERSONAL DATA MANAGEMENT FILE-IT 2™

This is Jerry White's popular database system for filing and managing personal and financial information. Create, store, manipulate and retrieve all types of data. Special financial entry and report programs create a powerful personal accounting system. Monthly Bar Graph program shows financial data on the screen and/or printer. Supports up to four disk drives as well as the Axlon Ramdisk (if you have one). Mailing list program generates mailing labels in one or two across format. Extensively documented in a ring binder. 24K Disk and Printer. \$49.95 + \$3.50 for shipping and handling.

FUN "n" GAMES #1™

WORDGAMES, POSSIBLE and LEAPFROG giving you hours of fun, challenge and entertainment. WORDGAMES, two games in one, contains GUESSIT - a deductive alphabetic reasoning game for one or two players and WORDJUMBLE - a multiple word scrambling puzzle with play-on-word hints and mystery answers. Instructions show how you can substitute your own words. Use POSSIBLE to help scramble word jumble puzzles or to create your own. All letter/number combinations or permutations of input are printed to screen or optional printer. LEAPFROG is a Chinese-Checker type jumping game in which you try to position two sets of animated jumping frogs in a minimum number of moves. 16K Cassette \$19.95; 24K Disk \$24.95. Disk version of GUESSIT works with VOTRAX Type "n" TALK. A real crowd pleaser.

COMING SOON! Space Shuttle Adventure Series™

Real-time Space Flight Simulations

COMING SOON! The Family Financier™

AN easy to use financial package.

ACCESSORIES

VINYL DUST COVERS

Custom sewn upholstery grade vinyl dust covers for ATARI 800 and 400 computers and the 810 and 825 peripherals. Protecting your equipment when not in use, these crisp black covers go well with any decor. Specify model(s) when ordering. \$14.95 each.

send check or money order including \$2.50 Shipping and Handling to:

SWIFTY SOFTWARE, INC.

64 BROAD HOLLOW ROAD
MELVILLE, N.Y. 11747
(516) 549-8141

N.Y. Residents add 7% sales tax

send for free catalogue dealer orders and c.o.d.'s accepted

©1981, 1982 Swifty Software, Inc.

NOTE: ATARI® is a registered trademark of Atari Inc., a Warner Communications Company and all references to ATARI® should be so noted.



Insight: Atari

Bill Wilkinson
Optimized Systems Software
Cupertino, CA

This month I will first respond to some of that unanswered mail; then part six of Inside Atari BASIC [a continuing series within this column] will delve further into string and array magic; and finally we will do a preliminary exploration of the depths of Atari's FMS.

Graphics Revisited

Actually, the title of this section might better be "machine language revisited." Probably none of my columns has generated as much response as part four of my Atari I/O series, subtitled "Graphics," in the February, 1982, issue of **COMPUTE!** Unfortunately, most of the response has been of the "I can't make it work" variety. Of course, my first response is "but I *know* it works!" Let still the letters ask, "How?"

I do not intend to turn this column into a tutorial on machine language. There are several good books available on 6502 machine language (including the Inmans' book specifically for *The Atari Assembler*), and any struggling beginner who is trying to make do without at least one of them is simply a masochist. However, my ego says that it will be better fed if more readers understand my articles.

For the most part, it seemed that those who had trouble with my February article assumed that what was published was some neat program to be used as is. *Not so!* I had simply given you a set of *subroutines* to use with your own programs. For an example, let us take a simple BASIC routine and its machine language equivalent. First, the BASIC:

```
30000 POKE 20,0 : POKE 19,0
30010 IF PEEK(19)=0 THEN 30010
30020 RETURN
```

Now, you would not mistake that for a complete BASIC program. But, if I told you that entering this routine and then executing a GOSUB 30000 from your program would produce a 4.2667-second pause, you would know when and how to use it. So let's do the same thing in machine language:

PAUSE

```
LDA #0
STA 20 ; "poke 20,0"
STA 19 ; "poke 19,0"
```

LOOP

```
LDA 19
BEQ LOOP ; "if peek(19)=0 then loop"
RTS      ; "return"
```

Again, this is *not* a complete program! But if you enter it (say at the end of your own machine language program) and then execute a JSR PAUSE, it will produce a 4.2667 second pause. Note, then, that JSR in machine language is the equivalent of GOSUB in BASIC.

The graphics routines (Program 5) in my February article are just subroutines, to be placed in your own machine language program and then JSRed to perform their actions. Perhaps the biggest mistake I made was in presenting these as an assembled listing (complete with " $\ast = \$660$ "). I certainly never used them as such. In point of fact, I tested them by .INCLUDEing them in my test programs, which were written with the OSS EASMD Assembler/Editor. And one of the test programs I used was, indeed, the example given on page 77 of that same article.

So how do *you* get these subroutines in and working for you? First and foremost, you obviously must type in all that code. Perhaps the best thing to do would be to type it in exactly as shown, including even the " $\ast = \$660$ " and the ".END". Then assemble it and carefully compare the object code generated with that in the magazine. When all appears correct, remove the " $\ast =$ " line and the ".END" line, renumber the whole thing (I would suggest REN 29000,5 or something similar), and LIST it to diskette or cassette. Now use NEW and write your mainline code. When you are reasonably satisfied with it, LIST it to disk or cassette also.

Now what? Obviously, if you have OS/A + , I suggest you use .INCLUDE (an assembler pseudo-op which allows you to include one file while assembling another). In fact, I tend to write assembly code structured as follows:

```
.INCLUDE #D:SYSEQU.ASM
<my mainline code>
.INCLUDE #D:library-routine-number-1
.INCLUDE #D:library-routine-number-2
...
.END
```

If my "mainline" code is big enough, I may even break it into two or three pieces and .INCLUDE each of them separately.

But what if you don't have .INCLUDE capability? Well, several assemblers have "FILE" or "CHAIN," which are not quite as flexible (since you don't return to where you left off after you have assembled a chained-to file...thus making the procedure next to useless for zero page equate files, etc.); but the principle is generally the same: put your mainline code first and then CHAIN to

the subroutine files.

And what if you have the Assembler/Editor cartridge? (For all of its faults, it is still a remarkably flexible tool, especially considering that it is usable with cassette-based systems.) Again, the principle holds. The only real difference is that you must do the INCLUDEs yourself. How? Via the ENTER command. If you haven't noticed it up until now, get your manual and read up on the "M" option of ENTER. You can merge two or more machine language program files (including cassette files) via the "M" option! Just as you can with BASIC, except that BASIC always presumes you want to merge.

Are there things to watch out for? Of course. Would I ever give you a method without a handful of caveats? (1) If you ENTER/merge a file with line numbers which match some (or all) of those in memory, you will overwrite the in-memory lines. (2) If you EVER forget the "M" option, you will wipe out everything in memory so far. (3) You won't find out about duplicate labels until you assemble the whole thing.

But even with all these cautions, I *strongly* recommend that you store each of your hard-earned routines on its own file/cassette. It then becomes almost easy to write the next program that needs some of those same routines.

By the way, caution number 1 in the previous paragraph is the reason I suggested RENumbering the graphics routines to 29000, or some such out of the way place. If you make notes of what each file (or cassette) does, as well as what line numbers it occupies, you can build a powerful library. And a P.S.: generally, INCLUDE, FILE, and CHAIN commands do not require unique line numbers, so you need not worry about RENumbering subroutines for use in such environments.

Gozinta and Gozouta

As long as we are on the subject of machine language techniques, I would like to point out the absolute necessity of establishing entry and exit conventions for each and every subroutine. Again, if you will refer to Program 5 from the February issue, you will note that each routine (GRAPHICS, COLOR, POSITION, PLOT, LOCATE, DRAWTO, and SETCOLOR) specified ENTER and EXIT conditions. For example, GRAPHICS requires that the desired graphics mode number be placed in the A-register before the JSR GRAPHICS. Upon return (RTS), the Y-register is guaranteed to contain a completion status.

On machines with more registers, it is good practice to write subroutines in a way that any registers not specifically designated in the ENTER and EXIT conditions are returned to the caller unchanged. On the 6502 microprocessor, though,

it is generally hard to write any significant routine that does not affect all three registers. Therefore, I have adopted the opposite convention for this CPU: If the ENTER/EXIT comments don't say otherwise, I presume that all registers are garbage when the routine returns. What convention you adopt doesn't really matter; just be sure to stick to one, and only one, method and you won't go wrong.

FILL From Machine Language

For those of you who are experienced machine language programmers and have not been kept entertained up to this point, take heart. The other question most asked about my February article was something like "so how do you call FILL from assembler?" I guess my comment that FILL from assembly language was exactly the same as from BASIC didn't make a very good impression. So, okay, I know when I'm licked. Herewith is a FILL subroutine, which I would hope you would include with the rest of the graphics routines and keep in your library for future use.

This time, I won't make the mistake of putting in line numbers and using "*"=" and ".END" This is a straight subroutine; type it in and JSR to it *only* after you have satisfied its ENTER conditions.

FILL H,V

```

: ENTER: Must have previously drawn the right hand edge
:         of the area to be FILLED via JSR's to PLOT and
:         DRAWTO. Just prior to JSR FILL, it must have
:         performed a JSR PLOT to establish the top (or
:         bottom of the line which will define the left edge of
:         the area to be FILLED. FILL presumes that the
:         color to fill with is that which was most recently
:         chosen via JSR COLOR. Finally, on entry, FILL
:         expects the registers to specify the ending position
:         of the line which will define the left edge of the
:         filled area, as follows:
:         h (horizontal) position in X,A registers
:           (X has LSB of position, A has MSB)
:         v (vertical) position in Y register
:
: EXIT:   Y-register has completion status from OS fill
:         routine

```

```

FILDAT = 765 ;where XIO wants the fill color
CFILL  = 18  ;fill is XIO 18
: rest of equates are from February article and program;
:
FILL

```

```

JSR POSITION ;subroutine from Feb. 1982
LDA SAVECOLOR ;value established via JSR
              COLOR
STA FILDAT ;see BASIC manual: color used
           for FILL
LDX #6*$10 ;file 6...where S: normally is

```



```

LDA #CFILL      ; the fill command (XIO 18)
STA ICCOM,X     ; ... is specified
LDA #0
STA ICAUX1,X    ; remember, XIO 18, #6, 0
JSR CIO         ; and let the OS do the work
RTS            ; ...and give us status in the Y-reg

```

By the way, did you notice that we didn't actually specify "S:" for the XIO, as specified in the BASIC manual? That's because the BASIC manual doesn't tell the whole truth. If you perform XIO on an already open file, the operating system ignores any filename you give it! Want to save a little space in your BASIC programs? Use 'XIO 18,#6,0,0,junk\$' where 'junk' is *any* string variable you happen to be using for any other purpose in your program.

Inside Atari BASIC: Part 6

Last month, we delved into the hopefully-no-longer-mysterious details on how string and array space is allocated from Atari BASIC and BASIC A+. We showed how to fool BASIC into believing that a perfectly ordinary string was located smack in the middle of screen space. The advantage of such deceptions is that BASIC can move strings of bytes at extremely high speeds, faster than you could ever hope to accomplish with any BASIC subroutine.

We did not discuss one other significant use of such string moves: Player/Missile Graphics. Obviously, if you can move the screen bytes around, you can move the players around just as well, and just as fast. Again, several games and utilities now available on the market use just this technique.

I also promised in the last column to tell of possible uses for multiple variables in the same address space (that is, having a string and an array occupying the same hunk of memory). If the idea interests you, read on.

One thing which BASICs in general lack is a good means of handling record input/output. How many times have you seen programs doing disk I/O using PRINT# and INPUT#? Yuch. (I have several reasons for that "yuch," but the best one is simply that PRINT#ing an item means that the number of disk bytes occupied depends upon the contents of the item.) But what is the alternative? With many BASICs, there is none. With Atari BASIC there is at least GET# and PUT#, but they are slow. So let us examine a way to make PRINT# and INPUT# work for us, instead of against us.

First, we will examine a small program:

```

100 DIM RECORD$(1),NAME$(20),QUANTITY
    ORDERED(0)
110 OPEN #1,8,0,"D:JUNK"

```

```

120 VVTP = PEEK(134) + 256*PEEK(135)
130 POKE VVTP+4,27 : POKE VVTP+6,27
140 GOSUB 900
150 PRINT "GIVE NAME AND QUANTITY:"
160 INPUT NAME$
170 INPUT TEMP : QUANTITYORDERED(0) =
    TEMP
180 PRINT #1;RECORD$
190 CLOSE #1
200 REM --- READ FILE WE JUST CREATED
    ---
210 OPEN #1,4,0,"D:JUNK"
220 GOSUB 900
230 INPUT #1,RECORD$
240 PRINT "WE READ BACK IN:"
250 PRINT ,,NAME$
260 PRINT ,,QUANTITYORDERED(0)
270 CLOSE #1
290 END
900 REM --- CLEAR THE VARIABLES ---
910 NAME$="                                ":REM
    20 BLANKS
920 QUANTITYORDERED(0)=0
930 RETURN

```

Surprised? Even though we cleared the variables in line 220, the input of line 230 re-read them from the file. How? Because line 130 set the dimension and length of RECORD\$ to 27, which includes the original single byte of RECORD\$, the 20 bytes of NAME\$, and the six bytes of the single element of the array QUANTITYORDERED. So PRINT# thought it had to print 27 bytes for RECORD\$, and INPUT# allowed RECORD\$ to accept up to 27 bytes.

Wow! With one fell swoop we have managed to allow fast disk I/O of any sized record, right? Wrong. Unfortunately, there are several limitations

to this technique. (1) The record cannot be over 255 bytes long or INPUT# won't be able to retrieve it all. And any size over 127 bytes will wipe out routines/data in the lower half of page \$600 memory. (2) The record cannot contain a RETURN (155 decimal, 9B hex) character. It will print fine, but the INPUT# will terminate on the first RETURN it sees. (3) The other strings in the record (NAME\$ in our example) will *not* have their lengths set properly by the INPUT#, thus necessitating something like the routine at line 900. But if you insert "280 PRINT LEN(NAME\$)", you will always get a result of 20.

Well, limitations one and three are easy enough to predict and understand, but how do you insure that your data does not contain a RETURN code? For strings which have been INPUT by a user, that's easy: the RETURN code will never appear in such a string. But what about numbers? Remember that we will be printing the internal form of Atari decimal floating point numbers. Can such numbers contain a byte with a value of 155 (\$9B)? Yes, but such a number would be in the range of -1E-74 to -9.E-73, which is unlikely enough to ignore for most purposes.

So, in summary, is this make-a-record technique useful? I'm not sure. Certainly BGET/BPUT or RGET/RPUT from BASIC A+ or their USR equivalents under Atari BASIC are much easier to code and use. And, yet, there is a certain elegance to record-oriented techniques which is not entirely lost to me. I probably will stick with the constructs we invented for BASIC A+, but I would respect a program using the above techniques.

A few last comments: the pokes of line 130 depend on RECORD\$ being the first variable defined. Recall my comments from last month about LISTing and reENTERing a program to insure a particular order of definition. Also, if you need to alter a variable other than variable number zero, remember that the formulas are:

VVTP+8 * VNUM+4 for the LSB of the length
VVTP+8 * VNUM+6 for the LSB of the DIMension

(and, again, see last month's article for fuller explanations).

And, finally, I really would be interested in hearing from anyone who uses the techniques I have devised here to produce a unique, real-world program that does things that can't be done otherwise.

Fun With FMS, Canto The First

Remember that fix for burst I/O I gave you in the May, 1982, issue? Did you try it? Did it prevent burst I/O errors? Yep. Did it slow down every kind of disk read? Yep. Oooooopsy daisy. Well, you can't be completely right all the time. This month,

we will try again.

First, I would like to explain, in terms of the FMS listing and the commentary (Chapter 12 – BURST I/O, *Inside Atari DOS*, **COMPUTE! Books**) why the fix I gave you in the May, 1982, issue worked insofar as it fixed the burst I/O problems.

To begin with, examine the code at locations \$09F8-\$09FD and \$0AD2-\$0AD7. These are the locations in PUT-BYTE and GET-BYTE, respectively, where the burst I/O routine is called. But lo! In PUT-BYTE, the JSR to burst I/O is directly preceded by a BCS, meaning that burst I/O won't occur unless carry is clear. But, in GET-BYTE, the JSR to burst I/O is directly preceded by a BCC – burst I/O occurs in read mode only if carry is set!

Now, if you examine the label "WTBUR" at \$0A1F, you will note that the first thing that occurs is a test of FCBFLG to find out if we are in update mode or not. If we are updating, we don't burst. But note that GET-BYTE called the label "RTBUR", AFTER the test, and so would always burst, whether in update mode or not. What I tried to do was change the "JSR RTBUR" (at \$0AD4) to a "JSR WTBUR" and then use the carry flag to distinguish between the type of request (I changed the BMI at \$0A24 to a BCC). *Great!* It worked! Except...it worked *too* well. Unfortunately, FCBFLG is zero (and therefore plus) when we have a file open for read only; so, therefore, the burst I/O was suppressed for *all* reads. Nuts.

We try again, using a slightly different approach. We will still count on the carry being set when called from PUT-BYTE and reset when called from GET-BYTE. This time, though, we will examine the actual I/O mode in use. FMS receives the I/O mode from CIO when the file is opened and places it in FCBOTC. Recall that the only legal values are 4, 6, 8, 9, and 12. Well, burst I/O is only illegal in modes 6 (read directory) and 12 (update). But mode 6 is handled separately (see \$0AC5-\$0ACB), so 12 is all we are really concerned with. Anyway, without further ado, here's the listing of the FMS patch:

*=\$0A1F

; first, patch the code where WTBUR used to be

; WTBUR
BURSTIO

LDA	FCBOTC,X	; Open Type Code byte
EOR	#\$0C	; check for mode 12... update
BEQ	NOBURST	; it IS update...don't burst
ROR	A	; move carry to MSB of A register
NOP		; filler only

TBURST

; ... and the STA BURTYP remains ... but now BURTYP is

```

; negative if BURSTIO was called from GET-BYTE and
; positive if it was called from PUT-BYTE.
;
      * = $0A41
; so we must patch here to account for the sense of being
; inverted from the original.
      BPL      WRBUR      ; called from PUT-BYTE
      * = $0AD4
; finally, we must patch the GET-BYTE call so that it no
; longer JSR's to RTBUR.
      JSR      BURSTIO    ; call the common burst
                           routine
;
      .END

```

And for those of you who don't want to type all that in, you might simply use BUG to do the following changes:

```

C A20 < 82,13,49,0C,F0,24,6A,EA
C A41 < 10
C AD5 < 1F

```

And, last but not least, from BASIC you may use the following:

```

POKE 2592,130
POKE 2593,19
POKE 2594,73
POKE 2595,12
POKE 2596,240
POKE 2597,36
POKE 2598,106
POKE 2599,234
POKE 2625,16
POKE 2773,13

```

Fun With FMS, Canto The Second

Not long ago, an OSS customer told me that he couldn't use Atari DOS to SAVE (option K on the menu) the contents of ROM. "How sneaky," cried I, "Best to use the SAVE command under OS/A +. We wouldn't do anything that nasty to you!"

But we did. And we do. And it isn't because we or Atari are sneaky or nasty; it is yet another phenomenon of burst I/O. Recall that when the burst I/O test is passed, FMS calls SIO to transfer the sectors of data directly from the user's buffer space. In order to do so, though, it must write the sector link information (last three physical bytes in a sector) into the correct spot in the user's buffer before calling SIO. Then, when SIO returns, it restores those three bytes and tries to write the next sector the same way. Again, if you have *Inside Atari DOS*, you can follow this happening at addresses \$0A52-\$0A7A, in the "WRBUR" code.

Ah...but what happens when you try to do burst I/O writes from ROM? FMS blindly tries to put its goodies into those three bytes and call SIO. SIO does what it is told, and FMS thinks that all is OK. Except that all is *not* OK! Those three bytes did *not* get changed, so what was written to the disk is garbage. And even ERasing the file won't work,

because the sector links are badly messed up. Crunchy, crunchy goes the disk, under worst-case circumstances.

Now this restriction is fairly easy to get around: one simply writes a program (in BASIC or machine language) which writes the desired bytes to the disk one at a time, thus preventing burst I/O. So I don't feel that I am giving away deep, dark Atari secrets when I give you an easier method to prevent burst I/O. Simply do either of the following:

```

from BUG:      C A2E < 0
from BASIC:    POKE 2606,0

```

Again, for those of you with the FMS listing, note that what we are doing is changing the AND #\$02 which checks for text mode (the read and write text line commands are \$05 and \$09, neither of which have bit \$02 turned on) into an AND #\$00 instruction, thus fooling the BEQ that follows into thinking that FMS can't do burst I/O because it's doing text mode I/O. Not too terribly tricky, and it works well.

I cannot recommend that you make this patch a permanent part of most system disks, since it completely disables burst I/O and makes the system load and save files considerably slower. Change it, use it, and then forget it.

ATARI OWNERS!

20% OFF ALL SOFTWARE

Adventure International * Avalon Hill *
Crystalware * Automated Simulations *
Arcade Plus * Gebelli Software * On Line
Systems * Horizon Simulations * IDSI *
Artworx * C.E. Software

Order from us and get:

Free Newsletter - Evaluations, Reviews!
No Club To Join - No Membership Fees!
Free Catalog!
Free Charge Card Use!
Free Phone Orders (we deduct the cost of the call)

COMPARE:

	Retail	Your Cost
Crush, Crumble, and Stomp	29.95	23.95
Jawbreaker	29.95	23.95
Mouseattack	34.95	27.95

To Order Call:
(412) 235-2970

Or Write:

MIDEASTERN SOFTWARE
Box 247 New Florence, PA 15944

Send money order, certified check, personal check (allow two weeks to clear), C.O.D. /

Add \$2.00 shipping per order
PA residents add 6% sales tax

Prices subject to change without notice.

Monthly Column

All computer users can benefit from this month's column — many of Bill's observations and hints are not specific to the Atari. If you're thinking of translating a BASIC game program into machine language to achieve greater speed, you'll find some valuable information below. For example, there's a discussion of the "ball/boarder" problem which can be the most difficult puzzle to solve when programming certain kinds of games.

Insight Atari

Bill Wilkinson
Optimized Systems Software
Cupertino, CA

This month we return to the world of program writing. As I noted in my last column, there has been a growing demand for me to explain how to write graphics programs in assembly language. So I will begin a two or three-part series this month on converting BASIC programs to assembly language. Although the programs will be specifically written for the Atari computers, it won't take too much imagination to convert them to Apple and Commodore machines.

The Bouncing BASIC Ball

Since we are going to try to build up this program in stages, we will start this month with the simplest possible form. Program 1 is an Atari BASIC program which bounces a "ball" around inside the rectangular screen. There is no scoring, no paddles, no sound, no players, no missiles, no intelligence.

In fact, perhaps the only thing which needs explaining is the frequent occurrence of the subexpression: $\text{INT}(n * \text{RND}(0))$. With Apple Integer BASIC, one could obtain the equivalent function by coding $\text{RND}(n)$; and I have often wished that Atari had let us include that capability in the original specifications for Atari BASIC (oh, well, maybe in the

Program 1. Simple Bouncing Ball Program

```
100 GRAPHICS 3
200 XMOVE=INT(5*RND(0))-2
300 YMOVE=INT(5*RND(0))-2
400 IF XMOVE+YMOVE=0 THEN 200
500 X=INT(40*RND(0))
600 Y=INT(20*RND(0))
700 XNEW=X:YNEW=Y
900 POKE 19,0:POKE 20,0:REM RESET TIMER
1000 REM LOOP STARTS HERE
1100 COLOR 0:PLOT X,Y
1200 COLOR 2:PLOT XNEW,YNEW
1300 X=XNEW:Y=YNEW
1400 XNEW=X+XMOVE:YNEW=Y+YMOVE
1500 IF XNEW<=0 OR XNEW>=39 THEN XMOVE=-XMOV
E
1600 IF XNEW<0 OR XNEW>39 THEN XNEW=X
1700 IF YNEW<=0 OR YNEW>=19 THEN YMOVE=-YMOV
E
1800 IF YNEW<0 OR YNEW>19 THEN YNEW=Y
1900 IF PEEK(19)=0 THEN 1000
2000 RUN
```

next version of BASIC A+?). Anyway, the idea is to produce an integer random number in the range of 0 to n-1, inclusive.

So now let's examine the program as a whole. (First, a comment: I have used the convention that X means "horizontal" and Y means "vertical." This

Program 2. Bouncing Ball Initialization

```
0000      20      .PAGE "initialization"
          30      ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
          40      ;
          50      ; A SIMPLE BOUNCING BALL PROGRAM
          60      ;
          70      ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
          80      ;

0000      90      * = $3000
          100     LINE100
          101     ;>>>

          GRAPHICS 3

3000 A903  0110     LDA #3
3002 20E830 0120     JSR GRAPHICS
          0197     ;
          0198     ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
          0199     ;
          0200     LINE200
          0201     ;>>>

          XMOVE=INT(5*RND(0))-2

3005 A904  0210     LDA #4
3007 205731 0220     JSR RND      ; GET RANDOM NUMBER FROM 0 TO 4
300A 38     0230     SEC
300B E902   0240     SBC #2      ; NOW IS RANDOM FROM -2 TO +2
300D 8DE230 0250     STA XMOVE   ; AS IN BASIC PROGRAM
          0297     ;
          0298     ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
          0299     ;
          0300     LINE300
          0301     ;>>>

          YMOVE=INT(5*RND(0))-2

3010 A904  0310     LDA #4
3012 205731 0320     JSR RND      ; GET RANDOM NUMBER FROM 0 TO 4
3015 38     0330     SEC
3016 E902   0340     SBC #2      ; NOW IS RANDOM FROM -2 TO +2
3018 8DE330 0350     STA YMOVE   ; AS IN BASIC PROGRAM
          0397     ;
          0398     ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
          0399     ;
          0400     LINE400
          0401     ;>>>
```


can have some strange implications. See below.) We start by establishing the least detailed graphics mode (which is, incidentally, roughly equivalent to Apple's LO-RES mode). Then we set both of the variables XMOVE and YMOVE to a random number in the range -2 to +2, inclusive. (Do you see how? 'INT(5*RND(0))' gives a number from zero to four, inclusive, and we then subtract two from it.) But we don't allow both values to be zero (line 400). (In a real "Pong" type game, you wouldn't want the X-motion to ever be zero. Here, allowing XMOVE to be zero is instructive.)

We then give the ball a starting position with X in the range of 0 to 39 and with Y from 0 to 19. Note that both the current position (X and Y) and the to-be-made-current position (XNEW and YNEW) are set equal. This is simply to get things started evenly. Line 900 resets the system timer. (You will have to do something differently here if you are using an Apple.)

The main loop is almost as simple. First, we erase (COLOR 0) the old "ball" (note that we are erasing nothing if this is the first time through the loop). Then we PLOT the new ball with a convenient, visible color (COLOR 2). We update our current ball position (line 1300) and also our to-be-made-current position (line 1400).

It Gets A Bit Difficult

Here is where it begins to get tricky. If the ball will be *at or beyond* the edge(s) of the screen, we must reverse its movement, as appropriate (lines 1500 and 1700). But suppose that the movement has *already* carried it beyond the screen bounds; we must then bring it back in-bounds (lines 1600 and 1800). Finally, for this simple demo,

```

                                IF XMOVE+YMOVE=0 THEN 200
301B ADE230 0410      LDA XMOVE
301E 18 0420      CLC
301F 6DE330 0430      ADC YMOVE      ; XMOVE + YMOVE
3022 F0E1 0440      BEQ LINE200      ; IF = 0 THEN 200
                                0497 ;
                                0498 ;
                                0499 ;
                                0500 LINE500
                                0501 ;>>>
                                X=INT(40*RND(0))
3024 A927 0510      LDA #39
3026 205731 0520     JSR RND      ; GET RANDOM NUMBER FROM 0 TO 39
3029 8DDE30 0530     STA X      ; AND KEEP IT
                                0597 ;
                                0598 ;
                                0599 ;
                                0600 LINE600
                                0601 ;>>>
                                Y=INT(20*RND(0))
302C A913 0610      LDA #19
302E 205731 0620     JSR RND      ; GET RANDOM NUMBER FROM 0 TO 19
3031 8DDF30 0630     STA Y      ; AND KEEP IT
                                0697 ;
                                0698 ;
                                0699 ;
                                0700 LINE700
                                0701 ;>>>
                                XNEW=X : YNEW=Y
3034 ADDE30 0710     LDA X
3037 8DE030 0720     STA XNEW      ; XNEW = X
303A ADDF30 0730     LDA Y
303D 8DE130 0740     STA YNEW      ; YNEW = Y
                                0897 ;
                                0898 ;
                                0899 ;
                                0900 LINE900
                                0901 ;>>>
                                POKE 19,0:POKE 20,0
3040 A900 0910      LDA #0
3042 8513 0920      STA 19
3044 8514 0930      STA 20      ; DON'T NEED TO DO LDA #0 TWICE
                                0997 ;
                                0998 ;
                                0999 ;
                                1000 LINE1000
                                1001 ;>>>
                                REM LOOP STARTS HERE
                                1097 ;
                                1098 ;
                                1099 ;
                                1100 LINE1100
                                1101 ;>>>
                                COLOR 0 : PLOT X,Y
3046 A900 1110      LDA #0
3048 201531 1120     JSR COLOR
304B AEDE30 1130     LDX X
304E ACDF30 1140     LDY Y      ; LOAD VALUES FOR SUBROUTINE CALL
3051 202031 1150     JSR PLOT
                                1197 ;
                                1198 ;
                                1199 ;
                                1200 LINE1200
                                1201 ;>>>
                                COLOR 2 : PLOT XNEW,YNEW
3054 A902 1210      LDA #2
3056 201531 1220     JSR COLOR
3059 A900 1230      LDA #0      ; (NEEDED FOR PLOT)
305B AEE030 1240     LDX XNEW
305E ACE130 1250     LDY YNEW
3061 202031 1260     JSR PLOT
                                1297 ;
                                1298 ;
                                1299 ;
                                1300 LINE1300

```


we simply do this loop until the clock ticks (4.26 seconds, roughly) and then start all over.

Even ignoring the limited goals of this program, there are a few significant flaws: (1) There is no visible border around the screen to tell you when and where the ball will "hit." (2) There are no sound effects. (3) The ball isn't round (or even remotely so). (4) Sometimes, the ball rebounds without hitting the wall. I am going to leave (1) and (2) for next time, and (3) can't really be changed without using player-missile graphics. But flaw (4) is an interesting one, and worth some discussion.

The problem lies in the basic algorithm I chose for moving the ball: the X and Y movements can range from -2 to +2 units, independently, and I move the ball each time in both X and Y according to the current movement factors (XMOVE and YMOVE). Let's take an example: suppose that the XMOVEment is zero and the YMOVEment is -2. And further suppose that the ball is currently at Y position +1 (one square from the edge of the screen). If I allow the ball to move to the new Y position determined by Y and YMOVE (YNEW = Y + YMOVE in line 1400), then it will be off the screen (YNEW will be -1). What to do?

One solution might be to pretend we have absorbent walls (IF YNEW < 0 THEN YNEW = 0). This will work, but will give strange flight paths for the ball. The solution I chose was to imagine that the ball hit the wall smack in the middle the two times I chose to make it visible. (Imagine: the ball is displayed at Y position +1. One-half of a time-tick later, it hits the wall and rebounds. Another one-half of a time-tick later, it has rebounded back out

```

1301 >>>
X=XNEW : Y=YNEW
3064 ADE030 1310 LDA XNEW
3067 BDE30 1320 STA X
306A ADE130 1330 LDA YNEW
306D BDDF30 1340 STA Y
1397 ;
1398 ;
1399 ;
1400 LINE1400
1401 >>>
XNEW=X+XMOVE : YNEW=Y+YMOVE
3070 ADDE30 1410 LDA X
3073 18 1420 CLC
3074 6DE230 1430 ADC XMOVE
3077 BDE030 1440 STA XNEW
307A ADDF30 1450 LDA Y
307D 18 1460 CLC
307E 6DE330 1470 ADC YMOVE
3081 BDE130 1480 STA YNEW
1497 ;
1498 ;
1499 ;
1500 LINE1500
1501 >>>
IF XNEW<=0 OR XNEW>=39 THEN XMOVE=-XMOVE
3084 ADE030 1510 LDA XNEW
3087 3006 1515 BMI THEN1500 ;XNEW < 0
3089 F004 1520 BEQ THEN1500 ;XNEW = 0
308B C927 1525 CMP #39
308D 9009 1530 BCC LINE1600 ;XNEW NOT >= 39
1550 THEN1500
308F A900 1555 LDA #0
3091 38 1560 SEC
3092 EDE230 1565 SBC XMOVE ;GET 0 - XMOVE
3095 BDE230 1570 STA XMOVE ; TO XMOVE
1597 ;
1598 ;
1599 ;
1600 LINE1600
1601 >>>
IF XNEW<0 OR XNEW>39 THEN XNEW=X
3098 ADE030 1610 LDA XNEW
309B 3006 1620 BMI THEN1600 ;XNEW < 0
309D C927 1630 CMP #39
309F F008 1640 BEQ LINE1700 ;XNEW = 39
30A1 9006 1650 BCC LINE1700
1660 THEN1600
30A3 ADDE30 1670 LDA X
30A6 BDE030 1680 STA XNEW ; XNEW = X
1697 ;
1698 ;
1699 ;
1700 LINE1700
1701 >>>
IF YNEW<=0 OR YNEW>=19 THEN YMOVE=-YMOVE
30A9 ADE130 1710 LDA YNEW
30AC 3006 1715 BMI THEN1700 ;YNEW < 0
30AE F004 1720 BEQ THEN1700 ;YNEW = 0
30B0 C913 1725 CMP #19
30B2 9009 1730 BCC LINE1800 ;YNEW NOT >= 19
1750 THEN1700
30B4 A900 1755 LDA #0
30B6 38 1760 SEC
30B7 EDE330 1765 SBC YMOVE ;GET 0 - YMOVE
30BA BDE330 1770 STA YMOVE ; TO YMOVE
1797 ;
1798 ;
1799 ;
1800 LINE1800
1801 >>>
IF YNEW<0 OR YNEW>39 THEN YNEW=Y
30BD ADE130 1810 LDA YNEW
30C0 3006 1820 BMI THEN1800 ;YNEW < 0
30C2 C913 1830 CMP #19
30C4 F008 1840 BEQ LINE1900 ;YNEW = 39
30C6 9006 1850 BCC LINE1900

```

to Y position + 1. We thus display it again at position + 1, since we are displaying only at integral time-ticks.) This choice is reflected in the programming in lines 1600 and 1800.

Of course, all "motion" via a computer is no more true motion than is a motion picture or a television picture. In truth, you are simply seeing a series of still pictures flashed in front of your eyes so quickly that your brain perceives the result as motion. Thus, there is nothing inherently wrong with my solution. Except that, from BASIC, the time between pictures is so long that even my lazy brain can sometimes clearly see that the ball didn't touch the wall. (Notice that if XMOVE is zero, so that we have only vertical ball movement, the effect is even easier to see.)

Can we do better? From BASIC, probably not. From assembly language, probably yes. If we choose a different algorithm, a different graphics mode, or make the pictures change faster, maybe we can give better illusions of motion. But that will wait for next time. This month, we will simply recode our BASIC routine in assembly language.

Having A Ball With Assembly Language

First note that the BASIC line numbers have been preserved, with line 100 in the assembly code having the label LINE100 and being followed, on line 101, with a remark containing the BASIC source for that line. (If you want to make your listings neat and readable, you might try the trick I used here: I placed a control-J [an ASCII line-feed character] both before and after the BASIC source. It can make your listing much more readable.)

Also note the inclusion of my graphics subroutines from the February issue of **COMPUTE!**

```

1860 THEN1800
30CB ADDF30 1870 LDA Y
30CB 8DE130 1880 STA YNEW ; YNEW = Y
1897 ;
1898 ;
1899 ;
1900 LINE1900
1901 ;>>>

IF FECK(19)=0 THEN 1000

30CE A513 1910 LDA 19
30D0 D009 1920 BNE LINE2000
1930 ;==== LEAVE THESE 6 LINES OUT FIRST TIME ====
30D2 A514 1940 LDA 20 ; LSB OF CLOCK
1950 CLOCKWAIT
30D4 C514 1960 CMP 20 ; CHANGED YET?
30D6 F0FC 1970 BEQ CLOCKWAIT ; NO...WAIT
1980 ;==== BUT ALWAYS KEEP LINE 1990 ====
30D8 4C4630 1990 JMP LINE1000
1997 ;
1998 ;
1999 ;
2000 LINE2000
2001 ;>>>

RUN

2010 ;
2020 ; to truly simulate RUN, we should store
2030 ; zero to all variables. For this program
2040 ; that is not necessary, since all variables
2050 ; are reset in the beginning of the program
2060 ; anyway. In general, though, be careful
2070 ; to check such things.
2080 ;
30DB 4C0030 2090 JMP LINE100
3000 ;
3010 ;
3020 ; !!! VARIABLES AND SUBROUTINES !!!
3030 ;
3040 ; There are no direct BASIC equivalents
3050 ; for the following lines ... the BASIC
3060 ; interpreter handles all this for you
3070 ; automatically.
3080 ;
3090 ;
3100 ;
3110 ; first -- the variable declarations
3120 ;
30DE 00 3130 X .BYTE 0
30DF 00 3140 Y .BYTE 0
30E0 00 3150 XNEW .BYTE 0
30E1 00 3160 YNEW .BYTE 0
30E2 00 3170 XMOVE .BYTE 0
30E3 00 3180 YMOVE .BYTE 0
3190 ;
3200 ; second -- the subroutines
3210 ;
3220 ; these are mostly the same subroutines that
3230 ; we presented in the February, 1982, issue
3240 ; though they have been modified slightly.
3250 ; Note that SETCOLOR and LOCATE have been
3260 ; dropped (not used in this program)
3270 ; and a new function, RND, has been added.
3280 ;
3290 ;
30E4 3300 .INCLUDE #D:GRAPHICS.ASM

Equates, etc., for graphics subroutines

30E4 9000 .PAGE "Equates, etc., for graphics subroutines"
9005 ;
9010 ; CIO EQUATES
9015 ;
E456 9020 CIO = $E456 ; Call OS thru here
0342 9025 ICCOM = $342 ; COMMAND to CIO in IoCb
0344 9030 ICBADR = $344 ; Buffer or filename Address
0348 9035 ICBLEN = $348 ; Buffer LENGTH
034A 9040 ICAUX1 = $34A ; AUXiliary byte # 1
034B 9045 ICAUX2 = $34B ; AUXiliary byte # 2
9050 ;
0003 9055 COPEN = 3 ; Command OPEN
000C 9060 CLOSE = 12 ; Command CLOSE

```


(Issue #21). I have added a RaNDom function, to make the mainline code easier and more compatible with the BASIC original. Even if you choose not to type in the mainline assembly language this month, you should type in and preserve these routines. Or simply add RND to the listing you typed in from February (you *did* type all that in, of course). We will use these same routines in the later articles in this series, but the listing will *not* be repeated.

As much as possible, the assembly language is self-explanatory, especially when coupled with the BASIC source. For example, what could be clearer than the translation of "GRAPHICS 3" into "LDA #3" and "JSR GRAPHICS"? If you don't understand *why* this works, you really need to get a good introductory book and read up on 6502 assembly language. For those of you into such things, you might note that when we convert from BASIC to assembly language, we tend to convert expressions by using reverse Polish notation. Thus, for example, line 300's assembly language equivalent might be expressed in "pidgin-HP" (that is, in a parody of the keyboard language used by HP reverse Polish calculators) as something like this:

4 RND 2 - ENTER xmove STORE
And those of you into FORTH will presumably also see the obvious corollaries.

The assembly language coding here is not the best nor the most efficient. For example, lines 410 through 430 could be replaced by a simple "ORA XMOVE" (because the A-register already contains YMOVE and because we don't really need the sum to find out if the two values are both zero). Rather, the idea here was to do as straightforward a translation as possible, allowing more of

```

000B 9065 CPBINR = 11 ; Command Put BINARY Record
0011 9070 CDRAW = 17 ; Command DRAWto
0075 ;
0004 9080 OPIN = 4 ; Open for INput
000B 9085 OPOUT = 8 ; Open for OUTput
0090 ;
0095 ;
0100 ; EQUATES used by the S: driver and
0105 ; the VBLANK routines
0110 ;
0055 9115 HORIZONTAL = $55
0054 9120 VERTICAL = $54
02FB 9125 DRAWCOLOR = $2FB
02C4 9130 COLOR0 = $2C4
0135 ;
0140 ; miscellany
0145 ;
00FF 9150 LOW = $FF
0100 9155 HIGH = $100
020A 9160 RANDOM = $D20A

```

The graphics subroutines

```

30E4 9165 .PAGE 'The graphics subroutines'
0170 ;
0175 ;
30E4 00 9180 SAVECOLOR .BYTE 0 ; where COLOR is saved
0185 ;
30E5 53 9190 SNAME .BYTE 'S:',0 ; the filename for open
30E6 3A
30E7 00

0195 ;
0200 ;
0205 ; GRAPHICS =
0210 ;
0215 ; ENTRY: A-res contains graphics mode 's'
0220 ; EXIT: Y-res has completion status
0225 ;
0230 GRAPHICS
30E8 48 9235 PHA ; save 's'
30E9 A260 9240 LDX #$10 ; file #
30EB A90C 9245 LDA #CCLOSE
30ED 9D4203 9250 STA ICCOM,X
30F0 2056E4 9255 JSR CIO ; First, we must close file #6
0260 ; (we ignore any errors from the close)
0265 ;
30F3 A260 9270 LDX #$10 ; again, file #6
30F5 A903 9275 LDA #COPN ; we will open this 'file'
30F7 9D4203 9280 STA ICCOM,X
30FA A9E5 9285 LDA #SNAME&LOW
30FC 9D4403 9290 STA ICBADR,X ; we use the file name 'S:'
30FF A930 9295 LDA #SNAME/HIGH
3101 9D4503 9300 STA ICBADR+1,X ; by pointing to it
0305 ;
0310 ; all is set up for OPEN, now
0315 ; we tell CIO (and S:) what kind of open
0320 ;
3104 68 9325 PLA ; our saved 's' graphics mode
3105 9D4B03 9330 STA ICAUX2,X ; is given to S:
0335 ; (note that S: ignores the upper bits of AUX2)
3108 29F0 9340 AND #$F0 ; now we set just the upper bits
310A 4910 9345 EOR #$10 ; and flip bit 4
0350 ; (Read the text. S: expects this bit inverted
0355 ; from what normal BASIC usage is.)
310C 090C 9360 ORA #$0C ; allow read and write access (f
or CIO)
310E 9D4A03 9365 STA ICAUX1,X ; make CIO and S: happy
3111 2056E4 9370 JSR CIO ; and do the OPEN of S:
3114 60 9375 RTS
0380 ;
0385 ;
0390 ;
0395 ; COLOR c
0400 ;
0405 ; ENTER: Color 'c' in A-register
0410 ; EXIT: Unchanged
0415 ;
0420 COLOR
3115 8DE430 9425 STA SAVECOLOR
311B 60 9430 RTS ; exciting, wasn't it?
0435 ;
0440 ;
0445 ;
0450 ; POSITION h,v

```

you to understand how simple assembly language can be.

Are there any tricky spots in the code? Not really. Though, if you are like me, you will have to pause each time you use a CMP and figure out if you really want BCS or BCC (or whether you also need a BEQ or...). Again, some of the CMP's could have been made simpler (for example, by using 'CMP #40' on line 1630 and omitting line 1640). And, again, I opted for consistency with the BASIC program.

The program does work. Try it. It took me about three hours to type it in and debug it (including about an hour of debugging the debugger). This represents much less time than it would have taken if I had not had the BASIC program as a working model. You might omit lines 1930 to 1980 the first time you run it. I won't tell you what will happen, but I will tell you that the lines are used to synchronize ball movement with the clock.

On Assembling And Debugging

You may have noted that the master origin ('*') for this program is at \$3000. If you use that origin and don't do anything special, assembling the program will wipe out the source code and *kablooey!* What can you do? Personally, I prefer to direct the object code to disk when I assemble. (I usually use 'ASM, #R:, #D:file.OBJ' where "file" is the same name as the source file and I use "R:" because I list to a DIABLO or DEC serial printer.) Then, with the source also safely LISTed to disk, I can use NEW and reLOAD the object and proceed to run and debug it. Using this method, it makes sense to place the origin somewhere fairly high in EASMD's (or the Assembler/Editor's) working

```

9455 ;
9460 ; ENTER: h (horizontal) position in X,A
9465 ; registers (LSB,MSB)
9470 ; v (vertical) position in Y-register
9475 ;
9480 ; EXIT: unchanged
9485 ;
9490 POSITION
3119 8655 9495 STX HORIZONTAL
311B 8556 9500 STA HORIZONTAL+1 ; read the text
311D 8454 9505 STY VERTICAL ; too simple, right?
311F 60 9510 RTS
9515 ;
9520 ;
9525 ;
9530 ; PLOT h,v
9535 ;
9540 ; ENTER: must have done a previous COLOR call
9545 ; X,A and Y registers set as in POSITION
9550 ;
9555 ; EXIT: Y-register has completion status
9560 ;
9565 PLOT
3120 201931 9570 JSR POSITION
3123 A260 9575 LDA #6*$10 ; file 6, again
3125 A90B 9580 LDA #CPBINR ; Command Put BINARY Record
3127 9D4203 9585 STA ICCOM,X
312A A900 9590 LDA #0
312C 9D4803 9595 STA ICBLN,X
312F 9D4903 9600 STA ICBLN+1,X ; if buffer length is zero...
3132 ADE430 9605 LDA SAVECOLOR ; then CPBINR puts one char from
A-res
3135 2056E4 9610 JSR CIO ; and this is how we PLOT
3138 60 9615 RTS
9620 ;
9625 ;
9630 ;
9635 ;
9640 ; DRAWTO h,v
9645 ;
9650 ; ENTER: must have done a previous PLOT
9655 ; X,A and Y registers as in POSITION
9660 ;
9665 ; EXIT: Y-register has completion code
9670 ;
9675 DRAWTO
3139 201931 9680 JSR POSITION
313C ADE430 9685 LDA SAVECOLOR
313F 8DFB02 9690 STA DRAWCOLOR ; where DRAWTO expects its color
3142 A260 9695 LDA #6*$10 ; file 6...once more
3144 A911 9700 LDA #CDRAW ; Just a command to 'S:'
3146 9D4203 9705 STA ICCOM,X
3149 A90C 9710 LDA #*OC
314B 9D4A03 9715 STA ICAUX1,X ; insurance
314E A900 9720 LDA #0
3150 9D4B03 9725 STA ICAUX2,X ; ...guaranteed to work
3153 2056E4 9730 JSR CIO ; do the actual DRAWTO
3156 60 9735 RTS
9740 ;
9745 ;
9750 ;
9755 ;
9760 ; RND r
9765 ;
9770 ; ENTER: r in A-register
9775 ; EXIT: A-register contains a random
9780 ; number from 0 to r, inclusive
9785 ;
9790 ;
9795 RND
3157 8D4631 9800 STA RTEMP
315A EE6631 9805 INC RTEMP ; makes CMP easier
9810 RNDWAIT
315D ADOAD2 9815 LDA RANDOM ; set a random number
3160 CD4631 9820 CMP RTEMP ; too big?
3163 B0F8 9825 BCS RNDWAIT ; yes...wait
3165 60 9830 RTS
9835 ;
9840 RTEMP .BYTE 0 ; RESERVE SPACE
9845 ;
9850 ;
9855 ;
3167 9999 .END

```


memory.

An alternative method is to keep the object code in memory *below* all my source listing. With EASMD this is easy to do. For example, with this program, I simply used a 'LOMEM 3800' command to tell EASMD not to use any memory below \$3800. With the Assembler/Editor cartridge, it is almost as easy: simply use BUG to issue "C2E5 < 00,38" and then "G A000". (\$02E5 is system LOMEM, which the Assembler picks up and uses for its own when it is coldstarted at \$A000.) In both instances, make sure you have LISTed off any program in memory before changing the LOMEM bound, since it is the occurrence of NEW which forces the change.

Actually, I often use *both* of the above measures. And even then I can run into problems. When I was working on this month's program, for example, I could assemble and then load the program fine. But when I went to use "G3000" from BUG, the system looped madly. I'm still trying to figure out why, but I solved it by loading the OBJect file from the operating system and then reentering the Assembler via a cold start. BUG then worked fine. I hope that by next month I will have figured out the reason for this strange behavior and will report a fix to you. (To be fair, I am using a *very* early pre-release version of the cartridge...perhaps you won't have this problem.)

Breakpoint Setting

Possibly the biggest fault of BUG (both versions) is the lack of easy breakpoint capabilities. Changing instructions to BRKs (\$00) and back often gets so tiresome that I tend to say the heck with it and try out an otherwise unchecked portion of code. When I'm lucky, it all works. When I'm not, I turn off the power and start again. Thank goodness I'm not trying to do this with just a cassette. The corollary? If you are using a cassette-only system, proceed with utmost caution and take the trouble to set lots of breakpoints.

That's about it for this month. Next month we will add several complications to the bouncing ball program. We will also explore some news, trivia, and gossip. And, whatever you do, don't believe everything that people say about the Atari and Atari BASIC: we may have some surprising benchmarks for you.

TOLL FREE
Subscription
Order Line
800-345-8112
In PA 800-662-2444

commodore
Check Our New
Lowest Prices!



PUBLICATIONS:

CBM User Guide	7 95
CBM Basic 4.0 Ref. Manual ..	9 95
CBM Disk Manual	7 95
CBM Printer Manual	7 95
MOS Hardware Manual	6 95
MOS Programming Manual	6 95
The PET Revealed	19 95
Library of PET Subroutines ..	19 95
Commodore Software Encyclopedia	9 95
CBM Programmer's Reference Manual	16 95

CBM EQUIPMENT:

CBM 4016 CPU (40 Col. Screen, 16K RAM)	790 00
CBM 4032 CPU (40 Col. Screen, 32K RAM)	990 00
CBM 8032 CPU (80 Col. Screen, 32K RAM)	1090 00
CBM 8096 CPU (80 Col. Screen, 96K RAM)	1590 00
CBM Micro Mainframe (Super PET)	1690 00
CBM 2031 Single Disk Drive (170K per 5 1/4 Diskette) ..	570 00
CBM 4040 Dual Disk Drive (170K per 5 1/4 Diskette) ..	990 00
CBM 8050 Dual Disk Drive (1 Meg per 5 1/4 Diskette) ..	1340 00
CBM 4022 Tractor Feed Printer	625 00
CBM C2N Cassette Deck (New Style)	65 00
CBM CPU/IEEE Cable	35 00
CBM IEEE/IEEE Cable	40 00
8023P Dot Matrix Printer (136 Col., 150 CPS)	790 00
Tally 8024-7 (7x7 Matrix) Printer	1275 00
Tally 8024-9 (7x9 Matrix) Printer	1425 00
8300P Letter Quality Printer (40 CPS)	1790 00

VIC EQUIPMENT:

VIC 20 (Includes RF Modulator)	255 00
VIC Single Disk Drive (170K per 5 1/4 Diskette)	470 00
VIC Joystick	9 95
VIC Modem	102 00
VIC 8K Memory Expander	59 95
VIC Super Expander	69 95
VIC 3K Memory Expander	39 95
VIC 2 Player Game Paddles	19 95
VIC 1515 Graphic Printer	325 00

VIC SOFTWARE:

VT 106A Recreation Six Pack (Cassette)	43 95
Includes Car Chase, Blue Meanies, Space Math, Slither/Super Slither, Biorythm Capability	
VT 107A Home Utility Six Pack (Cassette)	43 95
Includes Personal Finance 1, Personal Finance II, VIC Typewriter, Expense Calendar, Loan & Mortgage Calculator, Home Inventory	
VIC 1901 Vic Super Alien (Cartridge)	29 95
VIC 1904 Super Lander (Cartridge)	29 95
VIC 1908 Draw Poker (Cartridge)	29 95
AMOK (Cassette)	18 95
VIC Avengers (Cartridge)	29 95
Snakman (Cassette)	18 95

CBM SOFTWARE:

Wordcraft 80 Wordprocessor ..	295 00
Word pro 4+ Wordprocessor ..	325 00
OZZ Data Base System	295 00
Visicalc	200 00
Tax Preparation System	590 00
Dow Jones Portfolio	115 00
The Manager	250 00

COMING SOON:

More VIC Peripherals and Software
CBM 8250 Dual Disk Drive
Data Acquisition and Control Devices
Ultimax
Commodore 64
More VIC Software: Golf,
Omega Race, Wizard of Wor

All Items Insured
COD - UPS
Prepaid Orders Shipped Free
In Stock Items Shipped Within 48 Hours
MASTERCARD OR VISA ADD 3%
GA RESIDENTS ADD 4% SALES TAX

MART

P.O. Box 77286
Atlanta, Ga. 30357

404-981-5939

CALL TOLL FREE

**"CALL ABOUT
SUPPLIES"**

Watch For Our New "800" Number
Call or Write For a Catalog
• Complete Catalog
• VIC Software Catalog

Insight: Atari

Bill Wilkinson
Optimized Systems Software
Cupertino, CA

In addition to a continuation of the game development that I started last month, this month's column will delve into the argument of what makes BASIC run, including a chip that makes Atari BASIC run better. But first ...

FMS And Burst I/O, Yet Again

Well, July's column was supposed to fix the mistakes I made in the May column. And then, lo and behold, I blew it again in July. On page 172 of issue 26 of **COMPUTE!** there is a listing of changes to be made to FMS to help burst I/O work properly in update mode. The assembly language listing and the BUG changes were correct. Unfortunately, the POKEs from BASIC had one typo (my fault). The last POKE read

POKE 2773,13 ... WRONG!

should be

POKE 2773,31 ... RIGHT!

Speed And BASIC

Personally, I have never been sure that it is necessary for an interpreted language (e.g., BASIC) to be fast. Typically, I choose to use an interpreter for ease of use and speed of debugging, for writing quickie little programs, and for creating utilities that can run at any speed at any time.

But an increasing number of people are trying to use BASIC for writing serious software, including games, utilities, and business applications. Now I maintain that the speed of BASIC is irrelevant when it is being used for utilities (who cares how fast a disk fixer-upper runs?) or business applications (the program is usually waiting for keyboard, printer, or disk I/O anyway). But for writing games and a certain category of other programs (e.g., sorts), speed is important. But then why use BASIC? Because it's the easiest language to use? Because it can be made fast enough? Because it's the only language the author knows?

Actually, those (and many others) are all valid reasons to choose BASIC, as long as the author doesn't expect more than BASIC is capable of delivering. So what is BASIC capable of delivering?

A lot of adequacy. After all, look at some of the very successful games that are written in BASIC (*Crush, Crumble, and Chomp* is the first one that comes to my mind). Or look at some games that should never have been written in BASIC and yet were (a lot come to mind, but I will refrain from naming any).

Certain other authors writing in another magazine have claimed that Atari BASIC is the slowest language ever created. My first impulse was to say, "Who cares? It is the easiest to use, and that's more important." But I simply couldn't take that statement lying down, as it were. After all, if Atari BASIC is such a snail, how come all these programs seem to work just fine?

So I armed myself with five different BASIC interpreters: Applesoft, Atari BASIC, Atari Micro-soft BASIC, BASIC A+, and Cromemco's 32K Structured BASIC. Now OSS produced three of these five BASICs, so it might seem that I am prejudiced. Well...maybe a lot, but not too much. Some comments follow on what I decided to try to do.

I wanted to use a benchmark program that would, to some degree, show the fundamental speed of each BASIC. But I also wanted to see what impact such things as constants, variable names, and multi-statement lines would have. Luckily, at just about this same time, I happened upon a benchmark (as yet unpublished) which showed Atari faster than Applesoft in a very simple program. "Oh ho!" says I. "How can this be? Atari is the slowest machine ever, say certain voices."

Anyway, I began experimenting with a small benchmark program, allowing various changes so that I could see the impact on speed. The most fundamental program was simply:

```
10 < start a timer >
20 A=0 : B=12345.6
30 A=A+1.234567
40 IF A < B THEN 30
50 < print time used >
```

Obviously, the intent of this program is to cause a loop to execute 10,000 times. But what can be changed that will significantly affect the execution time without materially altering the program? Below I show all the versions of lines 20 and 30 that I tested. (Line 40 is not shown, but it followed line 20 in the naming of variables and otherwise remained unchanged.) The table also shows the times for the various languages, rounded to three significant figures.

In addition to the timings shown in Table 1, I also tried adding several variables to the programs. Adding 18 variables (in lines 11 and 12) added about five or six seconds to the Microsoft BASICs, about 1.5 seconds for Atari BASIC and BASIC

Table 1. The Speed Matrix

Lines 20 and 30

		Cromemco 32K BASIC	Atari BASIC	Atari uSoft	BASIC A+ (Atari)	Applesoft
A	20 A=0:B=12345.6 30 A=A+1.234567	37.0	72.6 (63.6)	270.	62.9 (59.3)	275.
B	20 A=0:B=12345.6 30 A=A+1.23456789	37.0	72.6 (63.6)	710.	62.9 (59.3)	350.
C	20 A=0:B=12345.6:C=1.234567 30 A=A+C	37.0	73.1 (64.1)	56.3	63.4 (59.8)	50.8
D	20 LONGVARIABLEA=0: LONGVARIABLEB=12345.6 30 LONGVARIABLEA= LONGVARIABLEA+1.234567	** 37.0	72.6 (63.6)	320.	62.9 (59.3)	can't do
E	20 LONGVARIABLEA=0: LONGVARIABLEB=12345.6 30 LONGVARIABLEA= LONGVARIABLEA+1.23456789	** 37.0	72.6 (63.6)	752.	62.9 (59.3)	can't do
F	20 LONGVARIABLEA=0: LONGVARIABLEB=12345.6: C=1.234567 30 LONGVARIABLEA= LONGVARIABLEA+C	** 37.0	73.1 (64.1)	106.	63.4 (59.8)	can't do

** These tests made using double precision variables in Cromemco BASIC and Atari Microsoft BASIC. Single precision times were shorter, but not significantly so.
() Times shown in parentheses are explained in the text.

A+, and nothing at all to Cromemco BASIC.

Also, I tried the effects of combining lines 30 and 40 into a single line. For example:

30 A=A+C: IF A<B THEN 30

The time savings were all in the area of one second, not surprisingly, so I have not detailed them here.

But, look at the surprises! Let's look at the "foreigner," Cromemco 32K BASIC, first. *Nothing* seems to make a difference to it! Actually, I knew that this would happen before I ran the tests. Of all the BASICs shown, Cromemco's is the most like a compiler. I simply included it to give you an idea of what a truly properly structured interpreter can accomplish, but we must be fair and admit that the language is 26K bytes in its smallest usable incarnation.

For you Atari BASIC and BASIC A+ programmers, the happiest surprise is perhaps simply finding out that these languages do as well as they do. Also, note that the various program changes have only a small effect on the running times. So you don't have to be too careful about how you write your programs. (But it is still true that putting subroutines and FOR/NEXT loops at the beginning of a program will make a noticeable speed difference. Don't feel too bad: all Microsoft BASICs

have this same quirk.)

And now to the Microsoft BASICs. Obviously, you pay a penalty for using constants in a loop. Using double precision constants (1.23456789 in our examples) costs so much that you should try to avoid them. Watch for long variable names: 41 seconds to go from a one-character name to LONGVARIABLEA? Ouch! (Actually, I also tried three-character names and found the penalty there to be over seven seconds.) And there is a penalty for having lots of variable names in use. Hmm... we need to use variable names instead of constants, because constants are so slow; but using lots of variable names costs time also, so...

How about the other side of the Microsoft coin? What can we do that will show off the Microsoft BASIC speed? Two answers: use integer variables and do some transcendental function calculations. It's reasonably obvious why integer variables help: integer arithmetic is guaranteed to take less time than floating point. But why the transcendental, if we just showed that the speeds are similar? Simple. I cheated. I used only addition, where the Atari BASIC floating point package shows up pretty good. But oh boy! Did we blow it when it comes to multiply! When using SIN, COS, etc., Atari Microsoft BASIC is three to six times faster

than Atari BASIC. Until now. But before I explain that "until," let me make a few points.

Microsoft BASIC is definitely capable of more speed than Atari BASIC, but *only* if you are *very* careful and use lots of programming tricks. If you are an advanced programmer, this won't bother you. But I still believe, as I did over three years ago when we designed Atari BASIC, that for most people (and especially for beginners and hackers like me) the ease of use that is the hallmark of Atari BASIC makes it a real standout. But of course I'm not the perfect, objective judge. So try all of the BASICs, if your budget can afford it, and judge for yourself.

Fast, Faster, Fastest

This section will explain that "until now" that I wrote in the next to last paragraph. As I said, we (OSS and predecessors) blew it when it came to implementing the multiply algorithm, and as a result the transcendental routines take long enough for you to go out and get a cup of coffee. *But...*

Newell Industries (alias Wes Newell) of 3340 Nottingham, Plano, Texas (75074) has introduced the *Fastchip*. Actually, the *Fastchip* is a ROM which replaces the OS Floating Point ROM in an Atari 400 or 800. Major portions of the 2K bytes of ROM have been changed, resulting in several speed and/or accuracy improvements. The biggest changes were to the multiplication (ta da!) routine and floating-point to integer conversion (which is used *all* the time, by GOTO, POKE, SETCOLOR, XIO, OPEN, and many, many other statements and functions).

I have said that I will not normally review software, but I think the *Fastchip* deserves an exception to this rule on two points: it *can* be considered hardware, and it is a must for anyone contemplating heavy math usage with an Atari. Just as an example, note the times in parentheses in Table 1. These times are those recorded with a *Fastchip* installed. And this in a benchmark which does *not* make heavy use of *Fastchip's* best features!

Newell Industries has done some fairly complete timings of the various routines, so I won't belabor that point here. I will, however, include my own small benchmark program, just to give you an idea of the improvements available.

As you will note, I have included the Microsoft timings, also. Quite frankly, comparing Microsoft with Atari BASIC in this benchmark is almost as ludicrous as the reverse comparisons in Table 1. Which perhaps says a lot about how worthwhile benchmark programs *really* are.

Anyway, note that using the *Fastchip* brings the Atari BASIC timings within striking range of the Microsoft timings. A *most* respectable perform-

Table 2. Transcendental Timings

line 30	Atari Microsoft	Atari BASIC	Atari BASIC with Fastchip
30 J = ABS(I)	1.15	1.53	1.48
30 J = SIN(I)	6.85	25.3	10.9
30 J = EXP(I)	6.75	33.7	9.93
30 J = I^I	12.4	74.0	20.8

```
10 <start timer>
20 FOR I=0 TO 6.3 STEP 0.02
30 J = <a function of I...see table>
40 NEXT I
50 <print elapsed time>
```

(program used with Table 2)

ance when you consider that the Atari BASIC routines use six byte floating point while Microsoft uses a four byte floating point. Incidentally, the BASIC A+ timings were all only a small fraction of a second faster than the Atari BASIC times here, so I omitted them.

Enough hard work. On with the games!

BOING ... Part 2

Last month, we started with a simple program to bounce a ball around in a box. We noted some problems having to do with bouncing fast balls against a wall when the "clock" is slow: either the ball hits the wall "invisibly" or the bounce has to look strange. This month, we will extend that program into a real game and present an alternative method of moving the ball.

If you did type in last month's program, you might try changing it so that you assign XMOVE and YMOVE instead of having the program pick random directions. I would suggest that you try values of 0, 0.5, 1.0, and 2.0 in various combinations. If you choose XMOVE = 1 and YMOVE = 0.5, you will accomplish roughly what this month's program will use. Note, though, that the ball appears to jerk across the screen in strange directions. If you slow down the movement loop (put a delay in it), you will see that the ball really does go in as straight a line as it can (given the coarseness of the display we chose, Graphics 3). The jerkiness is simply an optical illusion, as far as I can tell, due to your eye expecting a certain movement and then being fooled.

The solution? Really, with finite pixel positioning, there is none. But you can greatly improve the situation by using a higher resolution graphics mode while retaining a relatively large ball: the

jumps in the higher resolution mode are small in comparison to the ball and so are not perceived as readily. With an Atari, the easiest way to accomplish this is with Player/Missile Graphics; but I will not delve into that in this series of articles since the subject has been covered so thoroughly and well elsewhere. Rather, the intent of these articles is simply to give beginners to graphics and/or assembly language a start in converting ideas from paper to BASIC to assembler.

This month, though, there simply isn't room or time to show and explain both the BASIC program and its assembly language counterpart. So the assembly language version will wait for next month, but I promise that it will be as closely related to this month's BASIC as last month's pair of programs were interrelated.

By the way, for those of you who simply want to play the game, just type it in as carefully as possible. Then simply RUN it for a two player Table Tennis-like game, using joysticks (not paddles – and, by the way, you must hold the joysticks turned 90 degrees left from normal position). For a one player game (not exciting, but a good demo), hold down the START key as you hit the RETURN key after typing RUN. And thus we start a skeletal explanation of how this program works.

What Makes BOING Ping?

First, note that YP(x) and SCORE(x) are simply the Y (vertical) paddle position of player "x" and a count of that same player's misses (x is 0 or 1, only). SINGLE is a flag set by examining the console switches which creates either a two player or one player game. LASTWIN is a -1 or +1 flag which indicates who scored the last point (we initialize it randomly).

At line 2000, the real work begins. In Graphics mode 3, we draw top and bottom boundaries and left and right paddles and print the current score. If this is a single person game, we overlay the right paddle with another wall. Also, in line 2060, we initialize each player's paddle position to 10, smack in the middle of each side. The ball is also initialized somewhere in the middle of the screen and given a starting shove.

At lines 2600 and 2700, we use my trick for reading the left and right joystick positions (this is the reason for turning the paddles), and we skip moving the paddle if the joystick is centered (and we never move the right paddle in a SINGLES game). The method of moving a paddle is sheer simplicity: since each paddle is three units high, we erase the pixel on one end and create a new one on the other end. Presto, the paddle is moved. Oh, yes, we update YP(x).

Then, at line 3000, we start moving the ball.

This is pretty much like last month, except that the XMOVE is always plus or minus one while the YMOVE is -1, -0.5, 0, +0.5, or +1. Note that if the ball won't hit something on its next move, it is because it will miss a paddle, so someone (HITP) will lose a point.

But if the ball is hit by a paddle, its YMOVE-ment is not determined by simple reflection. Rather, if the ball hits the center of a paddle, it is reflected straight across the playing field (with YMOVE=0). If it hits directly on either side of center, it returns at a slight angle (YMOVE = -0.5 or +0.5). But if it just barely hits the edge of the paddle, it rebounds at a satisfactorily nasty angle (YMOVE = -1 or +1). All this is done in line 3080.

Finally, the "LOSE" and "SCORE" routines are fairly simple. We force the ball to continue its flight for two more steps and then make a nasty noise and a simple but flashy display. We award a hit point as appropriate and figure out who LASTWIN should be.

This is *not* a sophisticated game. It is *not* intended to awe you with the power and flexibility of the Atari computer. It is intended to be a simple enough game that most of you will be able to follow its logic. And it certainly is intended to be easily translated to assembly language. But that's next month.

```

1000 REM *** STARTUP THE GAME ***
1010 DIM YP (1),SCORE(1):SCORE(0)=0:SCORE(1)=0
1020 SINGLE=(PEEK(53279)<>7)
1100 LASTWIN=1:IF RND (0)>=0.5 THEN LASTWIN=-LASTWIN
2000 REM *** PREPARE FOR A SERVE ***
2010 GRAPHICS 3: COLOR 2: PLOT 0,0:DRAWTO 39,0
2020 PLOT 0,19:DRAWTO 39,19
2030 PRINT :PRINT SCORE(1),,SCORE(0):PRINT "
      SCORE";
2035 IF SCORE(0)>20 OR SCORE(1)>20 THEN END
2040 COLOR 3 :PLOT 0,9:DRAWTO 0,11:PLOT 39,9:DRAWTO 39,11
2050 IF SINGLE THEN COLOR 2:PLOT 39,0:DRAWTO 39,19
2060 YP(0)=10:YP(1)=10:REM VERTICAL POSITION
2070 IF SINGLE THEN LASTWIN=1
2100 REM SET UP BALL
2110 XMOVE=LASTWIN:YMOVE=INT(3*RND(0))--1:Y=INT(12*RND(0))+4
2120 YNEW=Y:X=19-5*XMOVE:XNEW=X
2500 REM *** MAIN PLAYING LOOP ***
2510 REM
2520 REM 1. CHECK AND MOVE PADDLES
2530 REM 2. SHOW NEW BALL POSITION
2540 REM 3. CHECK FOR COLLISIONS, ETC.
2550 REM
2590 REM *** FIRST CHECK AND MOVE PADDLES
2600 V0=PTRIG(0)-PTRIG(1):IF NOT V0 THEN 2700
2610 VP0=YP(0)-V0:IF VP0<2 OR VP0>17 THEN 2700
2620 COLOR 0:PLOT 0,YP(0)+V0:COLOR 3:PLOT 0,VP0-V0:YP(0)=VP0
2700 V1=PTRIG(2)-PTRIG(3):IF SINGLE OR V1=0 THEN 3000
2710 VP1=YP(1)-V1:IF VP1<2 OR VP1>17 THEN 3000
2720 COLOR 0:PLOT 39,YP(1)+V1:COLOR 3:PLOT 39,V

```

```

P1-V1:YP(1)=VP1
3000 REM *** BALL CONTROL ***
3010 COLOR 0:PLOT X,Y
3020 COLOR 1:PLOT XNEW,YNEW
3030 X=XNEW:Y=YNEW
3040 XNEW=XNEW+XMOVE:YNEW=YNEW+YMOVE
3050 IF XNEW<38 AND XNEW>1 THEN 3200
3060 HITP=(XNEW>20):XHIT=39*HITP
3070 IF SINGLE THEN IF HITP THEN 3100
3080 YMSAVE=YMOVE:YNEW=INT(YNEW):YMOVE=(YNEW-YP
(HITP))/2
3090 IF ABS(YMOVE)>1 THEN GOTO 4000
3100 XMOVE=-XMOVE
3200 IF YNEW=1 OR YNEW=18 THEN YMOVE=-YMOVE
3290 GOTO 2600
4000 REM *** THE 'LOSE' ROUTINE
4010 COLOR 0:PLOT X,Y
4020 COLOR 1:PLOT XNEW,YNEW
4030 FOR I=1 TO 10:NEXT I
4040 COLOR 0:PLOT XNEW,YNEW
4050 COLOR 2:PLOT XNEW+XMOVE,YNEW+YMSAVE
4130 SOUND0,132,12,12:POKE 20,0
4140 SETCOLOR 1,0,PEEK(20)*4:IF PEEK(20)<32 TH
EN 4140
4150 SOUND 0,0,0,0
4200 REM *** SCORE IT ***
4210 SCORE(HITP)=SCORE(HITP)+1
4220 LASTWIN=1:IF HITP THEN LASTWIN=--LASLTWIN
4990 GOTO 2000

```

COMPUTE! The Resource.

NEW FOR ATARI

FROM
MMG MICRO SOFTWARE

***** GAMES *****

CHOMPER - All machine language arcade game. One of the only computer games with intelligent monsters. Destined to become a classic.

Requires 16K, 1 Joystick · Disk or Cassette \$29.95

ASTEROID MINERS - Race against time! Your mission is to mine all the asteroids before time runs out. BASIC and machine language.

Requires 16K, 1 Joystick · Disk or Cassette \$29.95

*****NECESSITIES*****

DISK COMMANDER - Just save this program on your BASIC disks and it will autoboot and automatically list all programs from the disk into your screen. Simply run any program by typing in a number.

Requires 16K, Disk Only \$24.95

BASIC COMMANDER - This all machine language program is an absolute requirement for ATARI BASIC programmers. Single keystroke DOS and BASIC commands, plus; AUTONUMBER, RENUMBER, BLOCKDELETE and much more!

Requires 16K, Disk Only \$34.95

RAM TEST - The most thorough and fastest memory test available for the ATARI. This all machine language program takes 4 min. to test 48K. It's the only program that tests the cartridge area of RAM. Good for new 400/800 computer owners and for testing new RAM boards.

Requires 8K · Disk or Cassette \$24.95

*****BUSINESS/HOME*****

MAILING LIST - Extremely fast BASIC and machine language program. Each data disk holds over 500 files. Sort on any of 6 fields at machine language speed. Use any size labels or envelopes.

Requires 48K, Disk Only \$39.95

Please send check or money order to:

MMG MICRO SOFTWARE

P.O. Box 131 • Marlboro, NJ 07746

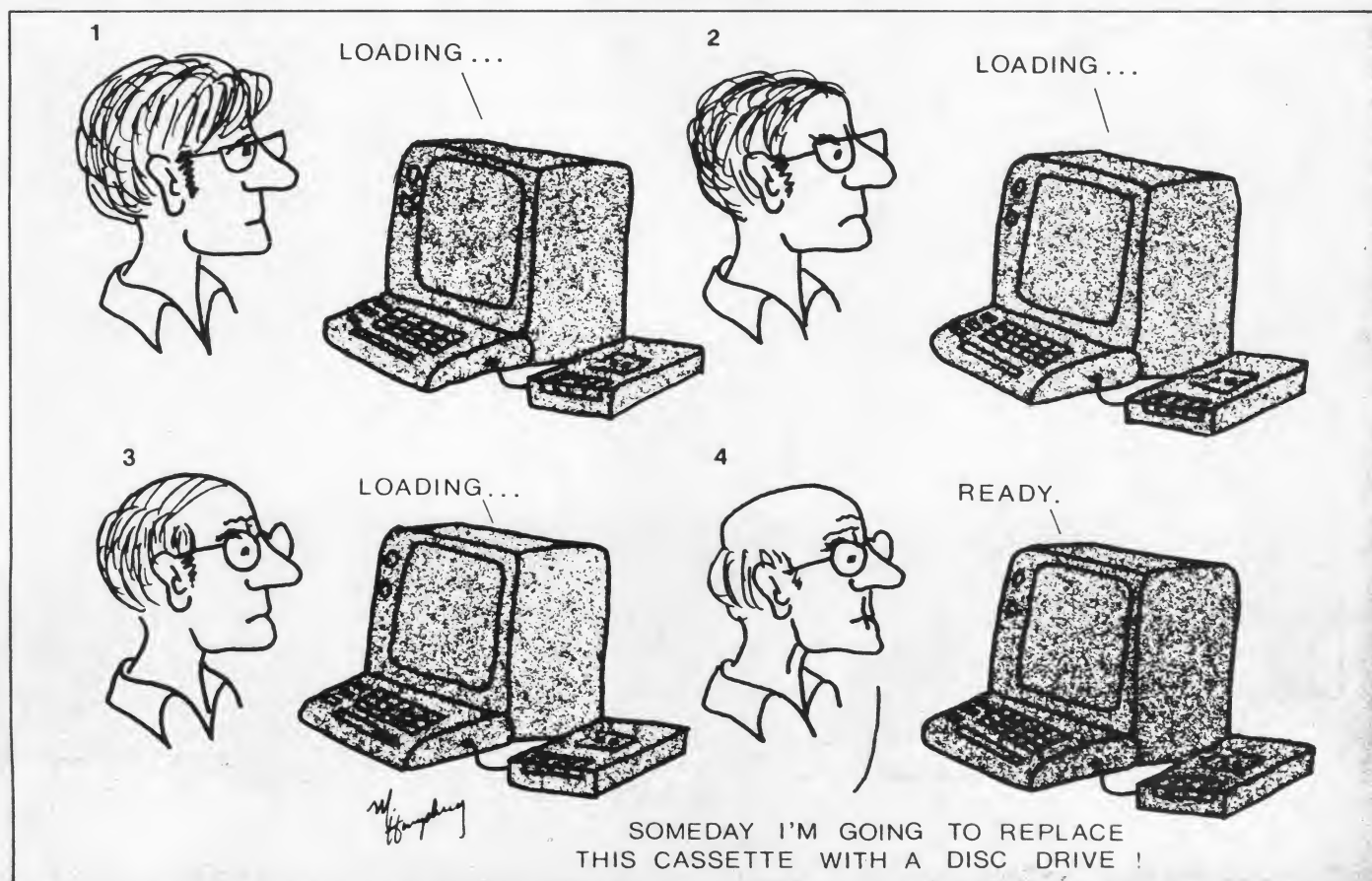
or call (201) 583-4362 for MasterCard, Visa or C.O.D.

Dealer and Distributor Inquiries Directed Exclusively To:

ProWare Systems

(201) 566-5007

ATARI is a registered trademark of ATARI, Inc.



A BASIC game is translated into machine language. The comments in the program will teach you how to PLOT, DRAWTO, COLOR, etc., in your own machine language games.

Insight: Atari

Bill Wilkinson
Optimized Systems Software
Cupertino, CA

Last month marked the first anniversary of this column in **COMPUTE!**, and I didn't even notice it. Which tells you how busy I am. We, like almost everyone in the software industry, are beginning to realize that survival comes only to those who diversify. So we are busily introducing new products and concepts. We think the net effect is beneficial to everyone: for us it means a chance to grow and try new approaches; for the user it means newer and better products with a wider choice than ever.

Of course, with the wider choice comes the obvious problem: which one of several competing packages should the user buy? I think I am asked that question only slightly less often than its predecessor: which computer should I buy? I usually sidestep the issue by saying something like this: "Find a software package that seems to do exactly what you want it to do. Ask for references from satisfied customers. When you are convinced that the software will suit your needs, buy the computer that is needed to run the particular software."

The most common problem I see is people buying too little computer for the problem they want to tackle. And, while the problem is sometimes related to the speed of the chosen machine (let's face it, you shouldn't be doing realtime voiceprint analysis with an Atari), the more common problem is simply lack of memory — both kinds of memory, RAM and disk.

This month, I have several topics of interest to Atari aficionados. And, of course, the monster listing of the assembly language version of the "Boing" game (the BASIC version was published last month). Please — hear my disclaimer: I am not nor do I claim to be a game programmer. I am quite aware that Boing is not the epitome of the gamer's art. Rather, I am here attempting to show the fundamentals of writing graphics games in assembly language. So don't type this game in expecting a miracle program; use it for instructional purposes only. Add to it, experiment with it, and chalk it up to experience.

A Boo-Boo

Well, so far we've encountered only one substantial mistake in our book, *Inside Atari DOS* (published by **COMPUTE!**). The error occurs in the text on page 11 and in the diagram (Figure 2-3) on page 14. Both correctly indicate the contents of the last three bytes of a data sector (the "link" information), but both assign the wrong order to these bytes. The byte containing the "number of bytes used in sector" is the *last* byte of the sector (byte 127 in single density sectors), *not* byte 125 as shown. Then the bytes shown as 126 and 127 move up to become 125 and 126, respectively.

Our apologies for the misinformation; we hope it didn't affect too many of you adversely. I think the mistake came about because of the comment in the listing at line 4312 on page 87, where the file number and sector link bytes are called "bytes 126, 127." Well, they are, if you are numbering from 1 to 128. The tables, etc., in the book are all numbered from 0 to 127; but recall that sectors on the disk are numbered from 1 to 720 (instead of 0 to 719). I don't know why we humans have such a hard time counting from zero, but we do. And computers have a hard time counting from any other number. Oh well.

Incidentally, the only other error in the diagrams that I have found occurs on page 21, where the labels "SABUFH" and "SABUFL" at the heads of the two columns are reversed.

CP/M For Atari?

I often get asked whether OS/A+ will run CP/M programs on the Atari (since externally OS/A+ looks very, very similar to CP/M — not an accident). But, you simply can't run CP/M on a 6502 (the heart of any Atari or Commodore or Apple). So how do Apple II owners run CP/M? Simple. They plug a card into their machine that essentially disables the 6502 and runs a Z-80 CPU instead. Why not do the same with an Atari?

First, let me say that I don't think that, as a practical matter, it is possible to *replace* the 6502 in the Atari 400/800 with another CPU (e.g., a Z-80). The reasons are many, but the primary one is the fact that the Atari peripheral chips (particularly Antic) seem somewhat permanently married to the 6502. However, there is no real reason that one could not put a co-processor board in the third slot of an 800 (the co-processor would probably have to have its own memory, though, to avoid interfering with the Atari's DMA and interrupt processing). This is essentially how some manufacturers have added 8086 capability to Apple II's. But it is expensive, since we now must pay not only for a CPU but also for 65K bytes of RAM and some sort of I/O to talk to the "main" 6502 CPU.

But doing this leaves you stuck with using the Atari serial bus to get data on and off a disk. And, aside from the slow speed, in my opinion an Atari 810 is really too small for practical CP/M work. So, what's the solution, if any? Actually, I've heard of a couple and know of one that is now working.

The first CP/M solution is to simply treat the Atari as an intelligent terminal and hook it up to a CP/M system. While this sounds like overkill, remember that most CP/M systems do not come with a terminal (screen and keyboard), and none can offer the color graphics capabilities of the Atari. But Vincent Cate (alias USS Enterprises) of San Jose, California, has come out with a hardware/software package that does more than make an Atari into an intelligent terminal. His package also allows most CP/M based computers with a 19,200 baud serial port to effectively replace the disk(s) and printer of an Atari computer.

The CP/M system is turned on and started up first, and it fools the Atari into believing that it is an 810 disk drive (just as does the 850 Interface Module in diskless systems). It thus boots a mini-pseudo-DOS into the Atari which simply passes file requests over the serial bus to the CP/M system. A great idea for someone who has a CP/M system and wants either to get a graphics terminal or to justify buying a game machine.

The primary limitation of this system is simply that you won't be able to read or write Atari-formatted diskettes, though it may be possible to CLOAD from an Atari cassette and then SAVE to the CP/M disk. You won't be compatible with the rest of the Atari world, but for games you probably don't care. At \$150, this is the cheapest CP/M to Atari connection, but it does presume the prior purchase of a CP/M-based system.

L. E. Systems (alias David and Sandy Small, et al.) has another method of doing co-processing: remove the cover of your 800 and replace it *and* the OS ROM board with an extension of the Atari's internal computer bus. On this bus one can stick more memory cards, disk controllers, and (of course) a Z80 card with its own 65K of memory. If your goal is to build a super powerful graphics machine, with access to the vast CP/M library, this is a workable approach (about \$1900 with two disk drives, *plus* the cost of the Atari 800).

However, for about the same money, you could buy a *real* CP/M machine (such as the Cromemco C-10) with 80-column screen, full function keyboard, built-in printer interface, bigger disks, etc. And then, if you wished, you could hook up your Atari via Vincent Cate's interface. The L. E. Systems' approach, though, assures lightning fast data and control flow between the Z80 and the 6502. More importantly, it allows you to con-

tinue to buy and use Atari-compatible disk-based software.

Finally, my rumor mill says that by the time you read this there will be a product available which will function as a more or less conventional Atari-compatible disk controller (à la Percom). But, at the flip of a switch, it will instead boot up and run CP/M (internal to the controller box), treating the Atari as an intelligent terminal, much as Vincent Cate's system does with more conventional CP/M computers.

Do I have any recommendations? Not really. Personally, I like my 128K Byte Cromemco (with 10 Megabyte hard disk and dual 1 Megabyte floppies) for serious software development. But when I think about it, I realize that the thing that makes this system so nice is *not* the CP/M compatibility (I almost never use CP/M, preferring to stick with Cromemco's Cromix). Rather, it is simply nice to have all that disk space available on command. So why get CP/M? Because you want to get into exotic compiler languages or because you need some very sophisticated business packages. Fine. But for games? Home finances? Learning how to program in BASIC? Graphics? I suggest you avoid CP/M.

Going With Boing

At last, we have here the complete listing of Boing as written in assembly language. As much as practicable, I have done a direct one-for-one translation from BASIC to machine code, without taking advantage of most of the foibles of the machine. Perhaps the only major change I have introduced is also the most unnoticeable from a casual reading of the source: I have made all the variables (which are six-byte floating point numbers in BASIC) into single bytes. This is *not* always possible. Sometimes, when writing in assembler, one needs numbers greater than 255; then one "simply" uses two-byte integers (or three or four-byte integers, or floating point even).

Except that, on a 6502, that "simply" isn't so simple. There are no 16-bit (or larger) instructions on a 6502, and one must simulate them using series of eight-bit loads, adds, stores, etc. For example, if this program were using Mode 8 graphics, where the horizontal position can vary from 0 to 319 (thus requiring a two-byte number to hold it), all of the code involving the "X..." variables would be larger and more complex. Lesson to be learned: use byte-size numbers whenever possible on a 6502.

Anyway, with regard to the listing of Boing, please note that I didn't leave enough space between my BASIC line numbers to allow my assembly language to share the numbering scheme. So I have put the BASIC lines into the listing in a way that makes them stand out for ease of reading.

Presuming that you have read my August and September columns, you will recognize the style and conversions that I have done. Statements such as PLOT, DRAWTO, COLOR, and others have been translated into JSRs to routines in my graphics package. (Note that the listing of the package has been omitted for space considerations. Simply include lines 9000 through 9999 of the listing in my August article.) I would, however, like to discuss a few points of interest.

Notice the coding of lines 2600 and 2700, where the BASIC program had used PTRIG(x)-PTRIG(x+1) to obtain a +1, 0, or -1 value from the joystick. But that requires turning the joystick 90 degrees from normal to play the game. As long as we are coding in assembly language, let's do it right!

What we have here, then, is essentially the code that BASIC A+ uses for its HSTICK(n) function. I think the code is easy to follow if you remember that the switches in the joystick force a zero bit in locations STICKn when they are pushed. By masking to only the bits we want, and by then inverting the bits, we are able to treat an "on" bit in a more or less normal fashion.

By the way, note that here, as elsewhere in the code, we are also using one-byte numbers to hold both positive and negative values. This works only so long as the absolute value of the signed numbers does not exceed 127, so be careful when using this technique.

Note the simulation of the array YP(n). First, look at how easy it is to handle array elements with constant subscripts, as in BASIC line 1010 (listing lines 1210 to 1230). Even variable subscripts aren't too hard when the array is byte sized and byte dimensioned. Look at BASIC line 4210 (listing lines 6030 and 6040). Admittedly, a true assembly language simulation of the BASIC line would probably go more like this:

```
LDX    HITP
LDA     SCORE,X
CLC
ADC     #1
LDX     HITP
STA     SCORE,X
        ; SCORE (HITP) = SCORE (HITP) + 1
```

But why not be a *little* smart when making conversions? Besides, if we were writing in some higher level languages, we could have written "INCREMENT SCORE(HITP)".

Finally, the hardest part of this conversion needs some analysis. As we noted last month, in order to provide better movement and bounce characteristics for the ball, we allowed it to have movements (and positions!) of -1, -0.5, 0, +0.5, and +1. But now we're in assembly language using

byte integers. How do we implement fractional movements? We can't really, so we must choose an equivalent scheme.

Notice the variables in the program called "Q.Yxxx". These variables all are used to hold values that represent *half* movements or positions. Example: if Q.YNEW contains 17, that means it is really representing position 8.5! Notice, then, that before plotting any point that is represented in this fashion, we must divide its value by 2 (by using a LSR instruction, c.f., listing lines 3820, 3930, etc.). Choosing this scheme has some interesting consequences: the last statement of BASIC line 3080 (listing lines 4500 through 4650) is, in some ways, the hardest part of this listing to understand, simply because of the implied "mixed-mode" arithmetic that is used. But it works!

Foibles Of The Assembler/Editor

Writing this article caused me to rediscover some of the foibles of the Atari Assembler/Editor cartridge (and EASMD, for that matter). For many of you, these quirks may seem normal, especially if you haven't used several different assemblers on various machines. But, to others, these eccentricities can be annoying or puzzling.

First, beware of the "*"=" pseudo-operator. It is *not* an origin operator ("ORG" in many assemblers), even though it is used as such! Any label associated with this pseudo-op will take on the value of the instruction counter *before* the operator is executed. This is necessary since "*"=" is *also* used to reserve storage ("DS" or "RMB" in some assemblers).

Examples:

```
LABEL1 *= *+5
        ; reserves five bytes of storage
        ; and assigns the label "LABEL1"
        ; to the five bytes
*= $4000
        ; sets the instruction counter
        ; to 4000 hex
LABEL2 *= $5000
        ; assuming this line followed one
        ; above, assigns 4000 hex to
        ; "LABEL2" and sets instruction
        ; counter to 5000 hex!
```

Second, examine any references to location "CLOCK.LSB" in the Boing listing (e.g., line 5870). Notice that, even though CLOCK.LSB is in zero page, the assembler produced a three-byte instruction for all references to it. This is because the *definition* of CLOCK.LSB did not occur until *after* the first *reference* to it! Actually, the assembler/editor is being remarkably clever here. Remember that the cartridge is, like most assemblers, a two-pass program. It reads the source once to determine where things are and will be, and then it reads the

source again to produce the listing and code. But, during the first pass through the source, it can't possibly know whether CLOCK.LSB is in zero page or not, so it chooses the safe route and assumes non-zero page. Then, lo and behold, it discovers that we really wanted the label to be in zero page. What to do?

If we now assign that label to zero page, the second pass of the assembler will produce only two bytes of code here, and all references to labels past that point will be off by one byte. We will have the infamous "phase error." So the assembler has a rule that states "once non-zero page, always non-zero page," and it continues to generate three-byte references. For a simple assembler like the Atari cartridge, this is a big step. It is still possible to produce phase errors with the cartridge, but it is more difficult than with many 6502 assemblers.

Third and last, there is a problem with the assembler/editor when it comes to multiple forward references. Consider the following code fragment:

```
AAA = BBB
BBB = CCC
CCC = 5
```

There is no way for a two-pass assembler to determine what the value of AAA is! On the first pass, it says "AAA is undefined, because BBB hasn't been defined yet." And then it thinks "BBB is undefined, similarly because of CCC." On the second pass, it *should* say "ERROR!! AAA is undefined, because BBB still hasn't been defined yet." But it can then produce "BBB is equal to 5 because that's what CCC is equal to."

Unfortunately, the assembler/editor doesn't keep a separate flag meaning "label as yet undefined." The "BBB = CCC" line is sufficient, from the assembler's viewpoint, to establish the existence of "BBB." So, on the second pass, it blindly puts the value of BBB (presumably zero) into AAA. Watch out for this trap! It has snared many a good programmer! I hope you realize that there would be no problems if you had coded that sequence in this order:

```
CCC = 5
BBB = CCC
AAA = BBB
```

That's it for this month. Next month we will investigate the many languages available to the Atari programmer. We will discuss and fix the major bug in Atari's 850 interface handler (the "Rn:" drivers). And maybe, just maybe, we will try to add cassette tape verification to BASIC.

```
1040 ;
1050 ;
1060 ;
1070 ; CAUTION: set memory origin according to
1080 ; your system needs!
1090 ;
1100 ;
1110 ;
0000 1120      *= $6000
      1130 ;
      1140 BOING
      1150 ;
```

```
:BASIC: 1010 DIM YP(1),SCORE(1):SCORE(0)=0:SCORE(1)=0
```

```
6000 4C0760 1160      JMP AROUND.DIM
6003 00      1170 YP      .BYTE 0,0      ; y-position
6004 00
6005 00      1180 SCORE  .BYTE 0,0      ; and score
6006 00
      1190 ;
      1200 AROUND.DIM
6007 A900 1210      LDA #0
6009 8D0560 1220      STA SCORE+0      ; SCORE(0)=0
600C 8D0660 1230      STA SCORE+1      ; SCORE(1)=0
      1240 ;
      1250 ;
```

```
:BASIC: 1020 SINGLE=PEEK(53279)<>7)
```

```
600F AD1F00 1260      LDA 53279      ; peek at console switches
6012 4907 1270      EOR #$07      ; A=7? Then A=0. A<>7? Then A<>0.
6014 8DE062 1280      STA SINGLE      ; set up our flag
      1290 ;
```

```
:BASIC: 1100 LASTWIN=1:IF RND(0)>=0.5 THEN LASTWIN=LASTWIN
```

```
6017 A001 1300      LDY #1      ; use y as temp for lastwin
6019 AD0AD2 1310      LDA RANDOM      ; get a random byte
601C 1002 1320      BPL HALFCHANCE
601E 88 1330      DEY      ; 50-50 chance that we do this
601F 88 1340      DEY      ; ...makes Y = $FF, same as -1
      1350 HALFCHANCE
6020 8CE162 1360      STY LASTWIN      ; store temp in final place
      1370 ;
```

```
:BASIC: 2000 REM prepare for a serve
```

```
1380 LINE2000
1390 ;
```

```
:BASIC: 2010 GR.3 : COLOR 2 : PLOT 0,0 : DRAWTO 39,0
```

```
6023 A903 1400      LDA #3
6025 20F362 1410      JSR GRAPHICS      ; GR.3
      1420 ;
6028 A902 1430      LDA #2
602A 202063 1440      JSR COLOR      ; COLOR 2
      1450 ;
602D A900 1460      LDA #0
602F A8 1470      TAY
6030 AA 1480      TAX
6031 202B63 1490      JSR PLOT      ; PLOT 0,0
      1500 ;
6034 A900 1510      LDA #0
6036 A227 1520      LDX #39
6038 A8 1530      TAY
6039 204463 1540      JSR DRAWTO      ; DRAWTO 39,0
      1550 ;
```

```
:BASIC: 2020 PLOT 0,19 : DRAWTO 39,19
```

```
603C A900 1560      LDA #0
603E AA 1570      TAX
603F A013 1580      LDY #19
6041 202B63 1590      JSR PLOT      ; PLOT 0,19
      1600 ;
6044 A900 1610      LDA #0
6046 A227 1620      LDX #39
6048 A013 1630      LDY #19
604A 204463 1640      JSR DRAWTO      ; DRAWTO 39,19
      1650 ;
```

```
:BASIC: 2030 .... NOTE: We don't print the scores in this version ....
```

```
1660 ;
1670 ;
```

```
0000 1000      .PAGE "      == GAME STARTUP ==
1010 ;
1020 ;
1030 ; This is the startup of BOING
```

```
:BASIC: 2040 COLOR 3:PLOT 0,9:DRAWTO 0,11:PLOT 39,9:DRAWTO 39,11
```

```
6003 1680 LDA #3
6004 2063 JSR COLOR ; COLOR 3
1700 ;
6052 A900 1710 LDA #0
6054 AA 1720 TAX
6055 A009 1730 LDY #9
6057 202B63 1740 JSR PLOT ; PLOT 0,9
1750 ;
605A A900 1760 LDA #0
605C AA 1770 TAX
605D A00B 1780 LDY #11
605F 204463 1790 JSR DRAWTO ; DRAWTO 0,11
1800 ;
6062 A900 1810 LDA #0
6064 A227 1820 LDX #39
6066 A009 1830 LDY #9
6068 202B63 1840 JSR PLOT ; PLOT 39,9
1850 ;
606B A900 1860 LDA #0
606D A227 1870 LDX #39
606F A00B 1880 LDY #11
6071 204463 1890 JSR DRAWTO ; DRAWTO 39,11
1900 ;
1910 ;
```

```
:BASIC: 2050 IF SINGLE THEN COLOR 2:PLOT 39,0:DRAWTO 39,19
```

```
6074 ADE062 1920 LDA SINGLE
6077 F016 1930 BEQ NOTTHEN2050 ; not single player mode
1940 ;
6079 A902 1950 LDA #2
607B 202063 1960 JSR COLOR ; COLOR 2
1970 ;
607E A227 1980 LDX #39
6080 A900 1990 LDA #0
6082 A8 2000 TAY
6083 202B63 2010 JSR PLOT ; PLOT 39,0
2020 ;
6086 A227 2030 LDX #39
6088 A013 2040 LDY #19
608A A900 2050 LDA #0
608C 204463 2060 JSR DRAWTO ; DRAWTO 39,19
2070 ;
2080 NOTTHEN2050
2090 ;
2100 ;
```

```
:BASIC: 2060 YP(0)=10:YP(1)=10
```

```
508F A90A 2110 LDA #10
5091 8D0360 2120 STA YP ; YP(0)=10
5094 8D0460 2130 STA YP+1 ; YP(1)=10
2140 ;
```

```
:BASIC: 2070 IF SINGLE THEN LASTWIN=1
```

```
5097 ADE062 2150 LDA SINGLE
509A F005 2160 BEQ LINE2100 ; NOT SINGLE
509C A901 2170 LDA #1
509E 8DE162 2180 STA LASTWIN ; LASTWIN=1 BECUZ SINGLE<0
2190 ;
```

```
:BASIC: 2100 REM SET UP BALL
```

```
2200 LINE2100
2210 ;
2220 ;
```

```
:BASIC: 2110 XMOVE=LASTWIN:YMOVE=INT(3*RND(0))-1:Y=INT(12*RND(0))+4
```

```
50A1 ADE162 2230 LDA LASTWIN
50A4 8DE362 2240 STA XMOVE ; XMOVE=LASTWIN
2250 ;
50A7 A902 2260 LDA #2
50A9 206263 2270 JSR RND ; get random number from 0 to 2
50AC 8DE662 2280 STA Q.YMOVE
50AF CEE662 2290 DEC Q.YMOVE ; then do the '-1'
50B2 0EE662 2300 ASL Q.YMOVE ; and convert to 'half-moves'
2310 ;
50B5 A90B 2320 LDA #11
50B7 206263 2330 JSR RND ; get random number from 0 to 11
50B9 2340 CLC
50BB 2350 ADC #4 ; '+4' as above
50BD 0A 2360 ASL A ; double number of moves to get
; half-moves
50BE 8DE562 2370 STA Q.Y ; Again, this is a 'half-position'
; variable
2380 ;
2390 ;
```

```
:BASIC: 2120 YNEW=Y : X=19-5*XMOVE:XNEW=X
```

```
60C1 ADE562 2400 LDA Q.Y
60C4 8DE762 2410 STA Q.YNEW ; YNEW=Y
2420 ; Here, we take advantage of the fact that XMOVE
2430 ; can only have values -1 or +1
60C7 A9FB 2440 LDA #0-5 ; assume XMOVE = +1
60C9 ACE362 2450 LDY XMOVE ; does XMOVE = +1?
60CC 1002 2460 BPL XMOVEPLUS ; yes
60CE A905 2470 LDA #5 ; no...so -5*-1 = +5
2480 XMOVEPLUS
60D0 18 2490 CLC
60D1 6913 2500 ADC #19 ; 19-5 OR 19+5
60D3 8DE262 2510 STA X
2520 ;
60D6 ADE262 2530 LDA X ; but you can see we don't really
; need this
60D9 8DE462 2540 STA XNEW ; XNEW = X
2550 ;
2560 ;
```

```
:BASIC: 2500 REM MAIN PLAYING LOOP
```

```
2570 ;
2580 ;
```

```
:BASIC: 2600 V0=PTRIG(0)-PTRIG(1):IF NOT V0 THEN 2700
```

```
2590 ; note that what we really want is V0=+1 if
2600 ; stick is pushed one way and V0=-1 if
2610 ; stick is pushed the other.
2620 ;
2630 LINE2600
60DC AD7802 2640 LDA STICK0 ; OS shadow location
60DF 2903 2650 AND #3 ; look at just fwd and backwd
; switches
60E1 4903 2660 EOR #3 ; invert the sense
60E3 F006 2670 BEQ GOTV0 ; if zero, stick not pushed
60E5 C901 2680 CMP #1 ; FWD pushed?
60E7 F002 2690 BEQ GOTV0 ; good...what we wanted
60E9 A9FF 2700 LDA #0-1 ; must be pulled back
2710 GOTV0
60EB 8DEB62 2720 STA V0 ; ta-da
2730 ;
60EE ADEB62 2740 LDA V0 ; so is stick pushed?
60F1 F03E 2750 BEQ LINE2700 ; IF NOT V0 THEN 2700
2760 ;
2770 ;
```

```
:BASIC: 2610 VP0=YP(0)-V0:IF VP0<2 OR VP0>17 THEN 2700
```

```
60F3 AD0360 2780 LDA YP+0 ; YP(0)
60F6 38 2790 SEC
60F7 EDEB62 2800 SBC V0
60FA 8DEB62 2810 STA VP0 ; VP0=YP(0)-V0
2820 ;
60FD C902 2830 CMP #2
60FF 9030 2840 BCC LINE2700 ; IF VP0<2 THEN 2700
6101 C912 2850 CMP #18
6103 B02C 2860 BCS LINE2700 ; or IF VP0>17 THEN 2700
2870 ;
2880 ;
```

```
:BASIC: 2620 COLOR 0:PLOT 0,YP(0)+V0:COLOR 3:PLOT 0,VP0-V0:YP(0)=VP0
```

```
6105 A900 2890 LDA #0
6107 202063 2900 JSR COLOR ; COLOR 0
2910 ;
610A AD0360 2920 LDA YP+0
610D 18 2930 CLC
610E 8DEB62 2940 ADC V0 ; YP(0)+V0
6111 A8 2950 TAY ; is y position
6112 A900 2960 LDA #0
6114 AA 2970 TAX
6115 202B63 2980 JSR PLOT ; PLOT 0,YP(0)+V0
2990 ;
6118 A903 3000 LDA #3
611A 202063 3010 JSR COLOR ; COLOR 3
3020 ;
611D ADED62 3030 LDA VP0
6120 38 3040 SEC
6121 EDEB62 3050 SBC V0
6124 A8 3060 TAY
6125 A900 3070 LDA #0
6127 AA 3080 TAX
6128 202B63 3090 JSR PLOT ; PLOT 0,VP0-V0
3100 ;
612B ADED62 3110 LDA VP0
612E 8D0360 3120 STA YP+0 ; YP(0)=VP0
3130 ;
3140 ;
```

:BASIC: 2700 V1=PTRIG(2)-PTRIG(3):IF SINGLE OR V1=0 THEN 3000

```

1310 LINE2700
1316 ; note that what we really want is V0=+1 if
1317 ; stick is pushed one way and V1=-1 if
1318 ; stick is pushed the other.
1319 ;
6131 AD7902 3200 LDA STICK1 ; OS shadow location
6134 2903 3210 AND #3 ; look at just fwd and backwd
; switches
6136 4903 3220 EOR #3 ; invert the sense
6138 F006 3230 BEQ GOTV1 ; if zero, stick not pushed
613A C901 3240 CMP #1 ; FWD pushed?
613C F002 3250 BEQ GOTV1 ; good...what we wanted
613E A9FF 3260 LDA #0-1 ; must be pulled back
;
6140 8DEC62 3270 GOTV1 STA V1 ; ta-da
;
6143 ADE062 3300 LDA SINGLE
6146 D045 3310 BNE LINE3000 ; IF SINGLE THEN 3000
6148 ADEC62 3320 LDA V1 ; so is stick pushed?
614B F040 3330 BEQ LINE3000 ; or IF V1=0 THEN 3000
;
;
;
3340 ;
3350 ;

```

:BASIC: 2710 VP1=YP(1)-V1:IF VP1<2 OR VP1>17 THEN 3000

```

614D AD0460 3360 LDA YP+1 ; YP(1)
6150 38 3370 SEC
6151 EDEC62 3380 SBC V1
6154 8DEE62 3390 STA VP1 ; VP1=YP(1)-V1
;
6157 C902 3410 CMP #2
6159 9032 3420 BCC LINE3000 ; IF VP1<2 THEN 3000
615B C912 3430 CMP #18
615D B02E 3440 BCS LINE3000 ; or IF VP1>17 THEN 3000
;
;
;
3450 ;
3460 ;

```

:BASIC: 2720 COLOR 0:PLOT 39,YP(1)+V1:COLOR 3:PLOT 39,VP1-V1:YP(1)=VP1

```

615F A900 3470 LDA #0
6161 202063 3480 JSR COLOR ; COLOR 0
;
6164 AD0460 3500 LDA YP+1
6167 18 3510 CLC
6168 6DEC62 3520 ADC V1 ; YP(1)+V1
616B A8 3530 TAY ; is y position
616C A900 3540 LDA #0
616E A227 3550 LDX #39
6170 202B63 3560 JSR PLOT ; PLOT 39,YP(1)+V1
;
6173 A903 3580 LDA #3
6175 202063 3590 JSR COLOR ; COLOR 3
;
6178 ADEE62 3610 LDA VP1
617B 38 3620 SEC
617C EDEC62 3630 SBC V1
617F A8 3640 TAY
6180 A900 3650 LDA #0
6182 A227 3660 LDX #39
6184 202B63 3670 JSR PLOT ; PLOT 39,VP1+V1
;
6187 ADEE62 3690 LDA VP1
618A 8D0460 3700 STA YP+1 ; YP(1)=VP1
;
;
;
3710 ;
3720 ;
3730 ;

```

:BASIC: 3000 REM *** BALL CONTROL ***

```

3740 LINE3000
3750 ;
3760 ;

```

:BASIC: 3010 COLOR 0 : PLOT X,Y

```

618D A900 3770 LDA #0
618F 202063 3780 JSR COLOR ; COLOR 0
;
6192 AEE262 3800 LDX X
6195 ADE562 3810 LDA Q.Y
6198 4A 3820 LSR A ; Divide half-position by 2 to
; get real pos'n
;
6199 A8 3830 TAY
619A A900 3840 LDA #0
619C 202B63 3850 JSR PLOT ; PLOT X,Y
;
;
;
3860 ;
3870 ;

```

:BASIC: 3020 COLOR 1:PLOT XNEW,YNEW

```

619F A901 3880 LDA #1
61A1 202063 3890 JSR COLOR ; COLOR 1
;
61A4 AEE462 3910 LDX XNEW
61A7 ADE762 3920 LDA Q.YNEW
61AA 4A 3930 LSR A ; Divide half-position by 2 to
; get real pos'n
;
61AB A8 3940 TAY
61AC A900 3950 LDA #0
61AE 202B63 3960 JSR PLOT ; PLOT XNEW,YNEW
;
;
;
3970 ;
3980 ;

```

:BASIC: 3030 X=XNEW:Y=YNEW

```

61B1 ADE462 3990 LDA XNEW
61B4 8DE262 4000 STA X ; X=XNEW
;
;
4010 ;
61B7 ADE762 4020 LDA Q.YNEW
61BA 8DE562 4030 STA Q.Y ; Y=YNEW
;
;
4040 ;
4050 ;

```

:BASIC: 3040 XNEW=XNEW+XMOVE:YNEW=YNEW+YMOVE

```

61BD ADE462 4060 LDA XNEW
61C0 18 4070 CLC
61C1 6DE362 4080 ADC XMOVE
61C4 8DE462 4090 STA XNEW ; XNEW=XNEW+XMOVE
;
;
4100 ;
61C7 ADE762 4110 LDA Q.YNEW
61CA 18 4120 CLC
61CB 6DE662 4130 ADC Q.YMOVE
61CE 8DE762 4140 STA Q.YNEW ; YNEW=YNEW+YMOVE
;
;
4150 ;
4160 ;

```

:BASIC: 3050 IF XNEW<38 AND XNEW>1 THEN 3200

```

;
4170 ;
61D1 ADE462 4180 LDA XNEW
61D4 C926 4190 CMP #38
61D6 B004 4200 BCS NOTTHEN3050
61D8 C902 4210 CMP #2
61DA B04C 4220 BCS LINE3200 ; XNEW<38 AND XNEW>1, SO GO
;
4230 ;
4240 NOTTHEN3050
;
4250 ;
4260 ;

```

:BASIC: 3060 HITP=(XNEW>20):XHIT=39*HITP

```

61DC A200 4270 LDX #0
61DE A000 4280 LDY #0
61E0 ADE462 4290 LDA XNEW
61E3 C914 4300 CMP #20 ; XNEW>20 ?
61E5 9004 4310 BCC XNEWLT20 ; NO
61E7 A001 4320 LDY #1 ; YES...SO 'TRUE' IS 1
61E9 A227 4330 LDX #39
;
4340 XNEWLT20
61EB 8CE962 4350 STY HITP
61EE 8EEA62 4360 STX XHIT
;
4370 ;
4380 ;

```

:BASIC: 3070 IF SINGLE THEN IF HITP THEN 3100

```

61F1 ADE062 4390 LDA SINGLE
61F4 F005 4400 BEQ LINE3080 ; NOT SINGLE
61F6 ADE962 4410 LDA HITP
61F9 D024 4420 BNE LINE3100 ; YES, SINGLE AND HITP
;
4430 ;
4440 ;

```

:BASIC: 3080 YMSAVE=YMOVE:YNEW=INT(YNEW):YMOVE=(YNEW-YP(HITP))/2

```

;
4450 ;
4460 LINE3080
61FB ADE662 4470 LDA Q.YMOVE
61FE 8DE862 4480 STA Q.YMSAVE ; YMSAVE=YMOVE
;
4490 ;
4500 ; REMEMBER: we are using half move increments in Q.Y...
4510 ; variables...so we really simply want to get
4520 ; rid of the lowest bit (the half step)
;
4530 ;
6201 ADE762 4540 LDA Q.YNEW

```



```

6204 29FE 4550 AND #SFE ; mask off last bit
        DE762 4560 STA Q.YNEW ; YNEW=INT(YNEW)
        4570 ;
6209 AEE962 4580 LDX HITP ; so X is either 0 or 1
620C 4A 4590 LSR A ; Q.YNEW / 2 gives the true YNEW
620D 38 4600 SEC
620E FD0360 4610 SBC YP,X ; YNEW-YP(HITP)
        4620 ; we don't need to divide by 2, because Q.YMOVE wants
        half-moves
6211 8DE662 4630 STA Q.YMOVE ; done
        4640 ;
        4650 ;

```

:BASIC: 3090 IF ABS(YMOVE)>1 THEN 4000

```

6214 ADE662 4660 LDA Q.YMOVE
6217 C903 4670 CMP #3 ; halfsteps, remember
6219 9004 4680 BCC LINE3100 ; 0,1, or 2 halfsteps
621B C9FE 4690 CMP #SFE
621D 902C 4700 BOC LINE4000 ; .aha...>2 halfsteps, <-2
        halfsteps
        4710 ;
        4720 ;

```

:BASIC: 3100 XMOVE=-XMOVE

```

        4730 LINE3100
621F A900 4740 LDA #0
6221 38 4750 SEC
6222 EDE362 4760 SBC XMOVE
6225 8DE362 4770 STA XMOVE ; xmove = -xmove
        4780 ;
        4790 ;

```

:BASIC: 3200 IF YNEW=1 OR YNEW=18 THEN YMOVE=-YMOVE

```

        4800 LINE3200
6228 ADE762 4810 LDA Q.YNEW
622B C902 4820 CMP #1+1 ; remember: half moves
622D F004 4830 BEQ THEN3200
622F C924 4840 CMP #18+18
6230 D009 4850 BNE NOTTHEN3200
        4860 ;
        4870 THEN3200
6233 A900 4880 LDA #0
6235 38 4890 SEC
6236 EDE662 4900 SBC Q.YMOVE ; 0-YMOVE
6239 8DE662 4910 STA Q.YMOVE ; is obviously the same as -YMOVE
        4920 ;
        4930 NOTTHEN3200
        4940 ;
        4950 ;

```

:BASIC: 3290 GOTO 2600

```

        4960 ;
        4970 ; if we simply jumped back to LINE2600 here, the game
        4980 ; would play impossibly fast...
        4990 ; so we put in a delay
        5000 ;
623C A900 5010 LDA #0
623E 8D1400 5020 STA CLOCK.LSB ; the 60th of a second ticker
        5030 DELAY1
6241 AD1400 5040 LDA CLOCK.LSB
6244 C902 5050 CMP #2 ; a 30th of a second?
6246 D0F9 5060 BNE DELAY1
        5070 ;
6248 4CDC60 5080 JMP LINE2600
        5090 ;
        5100 ;

```

:BASIC: 4000 REM *** the LOSE routine ***

```

        5110 LINE4000
        5120 ;
        5130 ; we will score the misses, even though we don't
        5140 ; display the results
        5150 ;
        5160 ;

```

:BASIC: 4010 COLOR 0:PLOT X,Y

```

624B A900 5170 LDA #0
624D 202063 5180 JSR COLOR ; COLOR 0
        5190 ;
6250 AEE262 5200 LDX X
6253 ADE562 5210 LDA Q.Y ; the half step

```

```

6256 4A 5220 LSR A ; becomes an integral step
6257 A8 5230 TAY
6258 A900 5240 LDA #0
625A 202B63 5250 JSR PLOT ; PLOT X,Y
        5260 ;
        5270 ;

```

:BASIC: 4020 COLOR 1:PLOT XNEW,YNEW

```

625D A901 5280 LDA #1
625F 202063 5290 JSR COLOR ; COLOR 1
        5300 ;
6262 AEE462 5310 LDX XNEW
6265 ADE762 5320 LDA Q.YNEW
6268 4A 5330 LSR A ; again, half step to full step
6269 A8 5340 TAY
626A A900 5350 LDA #0
626C 202B63 5360 JSR PLOT ; PLOT XNEW,YNEW
        5370 ;
        5380 ;

```

:BASIC: 4030 FOR I=1 TO 10:NEXT I

```

        5390 ; shoddy, shoddy. — using a for/next loop for timing!
        5400 ;
        5410 ; here, we do it right
626F A900 5420 LDA #0
6271 8D1400 5430 STA CLOCK.LSB
        5440 ;
        5450 DELAY2
6274 AD1400 5460 LDA CLOCK.LSB
6277 C902 5470 CMP #2 ; tick tock yet?
6279 D0F9 5480 BNE DELAY2 ; nope, maybe just tick
        5490 ;
        5500 ;

```

:BASIC: 4040 COLOR 0:PLOT XNEW,YNEW

```

627B A900 5510 LDA #0
627D 202063 5520 JSR COLOR
        5530 ;
6280 AEE462 5540 LDX XNEW
6283 ADE762 5550 LDA Q.YNEW ; starting to look familiar?
6286 4A 5560 LSR A
6287 A8 5570 TAY
6288 A900 5580 LDA #0
628A 202B63 5590 JSR PLOT ; PLOT XNEW,YNEW
        5600 ;
        5610 ;

```

:BASIC: 4050 COLOR 2:PLOT XNEW+XMOVE,YNEW+YMSAVE

```

628D A902 5620 LDA #2
628F 202063 5630 JSR COLOR ; COLOR 2
        5640 ;
6292 ADE462 5650 LDA XNEW
6295 18 5660 CLC
6296 GDE362 5670 ADC XMOVE ; x register = XNEW+XMOVE
6299 AA 5680 TAX
629A ADE762 5690 LDA Q.YNEW
629D 18 5700 CLC
629E GDE862 5710 ADC Q.YMSAVE
62A1 4A 5720 LSR A ; integerize the sum
62A2 A8 5730 TAY ; y register = YNEW+YMSAVE
62A3 A900 5740 LDA #0
62A5 202B63 5750 JSR PLOT ; PLOT it
        5760 ;
        5770 ;

```

:BASIC: 4130 SOUND 0,132,12,12:POKE 20,0

```

62A8 A984 5780 LDA #132
62AA 8D00D2 5790 STA SOUND.FREQ ; implicitly channel 0
62AD A9CC 5800 LDA #12*16+12
62AF 8D01D2 5810 STA SOUND.CONTROL ; ,12,12 also for channel 0
62B2 A900 5820 LDA #0
62B4 8D1400 5830 STA CLOCK.LSB ; finally, BASIC did it right!
        5840 ;
        5850 ;

```

:BASIC: 4140 SETCOLOR 1,0,PEEK(20)*4:IF PEEK(20)<32 THEN 4140

```

        5860 LINE4140
62B7 AD1400 5870 LDA CLOCK.LSB ; same as PEEK(20)
62BA 0A 5880 ASL A
62BB 0A 5890 ASL A ; * 4
62BC 8DC502 5900 STA SETCOLOR1 ; control register number 1
        5910 ;

```

```

62BF C980 5920 CMP #32*4 ; a little tricky...can you
                                follow it?
62C1 90F4 5930 BOC LINE4140 ; it works...really
                                5940 ;
                                5950 ;

```

```

:BASIC: 4150 SOUND 0,0,0,0

```

```

62C3 A900 5960 LDA #0
62C5 8D00D2 5970 STA SOUND.FREQ
62C8 8D01D2 5980 STA SOUND.CONTROL
                                5990 ;
                                6000 ;

```

```

:BASIC: 4200 REM *** SCORE IT ***

```

```

                                6010 ;
                                6020 ;

```

```

:BASIC: 4210 SCORE(HITP)=SCORE(HITP)+1

```

```

62CB AEE962 6030 LDX HITP
62CE FE0560 6040 INC SCORE,X ; isn't assembler easy?
                                6050 ;
                                6060 ;

```

```

:BASIC: 4220 LASTWIN=1 : IF HITP THEN LASTWIN=LASTWIN

```

```

62D1 A901 6070 LDA #1
62D3 AEE962 6080 LDX HITP ; if HITP?
62D6 F002 6090 BEQ NOT.HITP ; no
62D8 A9FF 6100 LDA #0-1 ; yes...so make it -1
                                6110 NOT.HITP
62DA 8DE162 6120 STA LASTWIN ; that's all that is needed
                                6130 ;
                                6140 ;

```

```

:BASIC: 4990 GOTO 2000

```

```

62DD 4C2360 6150 ;
                                6160 JMP LINE2000
                                6170 ;

```

BOING — not quite up to PONG
GENERAL RAM USAGE

```

62E0 6180 .PAGE "GENERAL RAM USAGE"
6190 ;
62E0 00 6200 SINGLE BRK ; flag for one-player game
62E1 00 6210 LASTWIN BRK ; who won last time?
                                6220 ;
                                6230 ; the x moves
                                6240 ;
62E2 00 6250 X BRK ; current x position
62E3 00 6260 XMOVE BRK ; current x movement
62E4 00 6270 XNEW BRK ; new x position
                                6280 ;
                                6290 ; and the y positions and moves
                                6300 ;
                                6310 ; remember: the Q.Ybox locations reference positions
                                6320 ; or movements in terms of half steps
                                6330 ;
62E5 00 6340 Q.Y BRK ; current y position
62E6 00 6350 Q.YMOVE BRK ; current y movement
62E7 00 6360 Q.YNEW BRK ; new y position
62E8 00 6370 Q.YMSAVE BRK ; saved for LOSE routine only
                                6380 ;
                                6390 ; other miscellany
                                6400 ;
62E9 00 6410 HITP BRK ; the HIT Person...who missed
62EA 00 6420 XHIT BRK ; where the miss occurred
                                (x position)
                                6430 ;
62EB 00 6440 V0 BRK ; just a temporary
62EC 00 6450 V1 BRK ; ditto
                                6460 ;
62ED 00 6470 VP0 BRK ; Vertical position of Paddle 0
62EE 00 6480 VP1 BRK ; Vertical position of Paddle 1
                                6490 ;
                                6500 ; system equates
                                6510 ;
0012 6520 CLOCK = 18 ; the system clock
0014 6530 CLOCK.LSB = CLOCK+2 ; the 60th of a second ticker
0278 6540 STICK0 = $278 ; OS shadow read of first stick
0279 6550 STICK1 = $279 ; ditto for second stick
D200 6560 SOUND.FREQ = $D200 ; port which controls channel
                                0 freq
D201 6570 SOUND.CONTROL = $D201 ; and control
                                6580 ;
02C5 6590 SETCOLOR1 = $2C5 ; also known as COLPF1

```

BOING — not quite up to PONG
The GRAPHICS subroutines

```

62EF 6600 .PAGE "The GRAPHICS subroutines"

```

```

6372 6630 .OPT LIST
6372 6640 .END

```

[Put the graphics subroutines from line 9000 on up (pg. 150, **COMPUTE!**, August 1982) here.]

ATARI™ 400/800 OWNERS!

Discover

DATAPORT

YOUR SOURCE FOR THE BEST IN ENTERTAINMENT
AND EDUCATIONAL SOFTWARE! OUR NEW
CATALOG LISTS PAGE AFTER PAGE OF GAMES ON
BOTH CASSETTE AND DISK! SEND \$1.00 NOW TO:

DATAPORT
P.O. BOX 975
NORTHBROOK, IL 60062

GAMES

FOR THE

ATARI 400/800

OVER 100 GAMES, SIMULATIONS,
ADVENTURES AND MORE!!

20% OFF LIST PRICE!

Artworx · Adventure International · On Line Systems · CE

Automated Simulations (EPYX) · Arcade Plus

Gebelli · Avalon Hill · Crystal · Broderbund · IDSI

Budgeo · Datasoft (and more!!)

FREE CATALOG, NEWSLETTER

TO ORDER CALL

(412) 235-2970

OR WRITE

mideastern software

BOX 247 NEW FLORENCE, PA. 15944

add \$2.00 shipping/handling per order
PA residents add 6% sales tax

A quick way to verify cassettes, a survey of languages available for the Atari, and a fix for a bug in Atari's RS-232 handlers.

Insight: Atari

Bill Wilkinson
Optimized Systems Software
Cupertino, CA

Well, I didn't quite make it. I was trying to have a cassette verify program done in time for this month's column, but the pressures of writing a couple of sections for the new *COMPUTE!'s Book of Atari Graphics*, producing three major new OSS products, and answering literally hundreds of phone calls got to me. So, wait for next month. But in the meantime, at least I have a quickie verify method that might keep the frustrations away for a month.

Quick And Dirty

One of the major flaws of the Atari computers has always been the lack of a cassette verify capability. But there is an almost effortless way to simulate this missing capability.

The secret lies in the fact that, because of Atari's superior operating system and because BASIC interfaces properly to it, you can LIST to any file or device. So, when you are ready to save your program to cassette, do *not* use CSAVE. Instead, Use LIST"C:" to produce an ATASCII listing on the cassette. Then you can rewind the tape and, *without deleting or changing the program in memory*, enter the following direct statements:

```
DIM Q$(256) : OPEN #1,4,0,"C:"
FOR Q = TO 100000:INPUT #1,Q$:PRINT Q$:
NEXT Q
```

Do you see the reason for the trick? Atari makes no distinction between a listing file and a data file, even on a cassette, so we can simply read the listing as data and print what we read on the screen. If what appears on the screen is correct, the cassette was recorded correctly. Incidentally, the FOR/NEXT loop is only needed so that we can enter the statements in direct mode (without line numbers). The loop will execute more times than there are lines in the file, but the end of file error will stop the process anyway. (And it is a good idea to type "END" after getting the end of file error.)

For the adventuresome, there might be an even easier way. After using the LIST"C:", simply rewind the tape and type ENTER"C:". Remember,

ENTER does *not* erase the program in memory, but instead merges the filed program with the current one. But in this case, since the two programs are the same (if the file was recorded correctly), the ENTER should have no visible effect. If there is an error in the tape, the ENTER will simply halt and no harm will be done. Theoretically. In truth, it is possible that one line could be destroyed (if it were partially ENTERed from one tape block and then blew up in the next block). I have not tested this exhaustively, so use it at your own risk.

Foreign Languages

What is *the* Atari language? What is the best language for doing the most things with an Atari computer? Is there such a thing? There may be no good answers to these questions, but trying to answer them may prove interesting, so let's give it a shot.

The Atari now has a respectable complement of languages available for it. I will list those I know of here and I must apologize in advance for any omissions. The listings within each category are roughly in order of date of introduction of the product. An asterisk indicates a product no longer actively advertised, so check with the publisher for availability.

Assemblers

Cassette-Based Assembler – Quality Software*
Assembler/Editor Cartridge – Atari, Inc.
EASMD (Edit/ASseMble/Debug) – OSS, Inc.*
DATASM/65 – Datasoft*

Macro Assemblers

MAE (Macro Assembler/Editor) – Eastern House
Macro Assembler – ELCOMP
AMAC (Atari MACro assembler) – Atari, Inc.
MAC/65 – OSS, Inc.

Interpreters

Atari BASIC – Atari, Inc.
BASIC A# – OSS, Inc.
LISP – Datasoft
PILOT – Atari, Inc.
tiny c – OSS, Inc.
Microsoft BASIC – Atari, Inc.

Pseudo-Compilers

QS FORTH – Quality Software
Atari FORTH – Atari Program Exchange
pns FORTH – Pink Noise Studios
PASCAL – Atari Program Exchange
ValForth – Valpar

Compilers

PASCAL – Atari Program Exchange
C/65 – OSS, Inc.

I admit I hesitated over classifying FORTH as a pseudo-compiled language, but I was trying to

group the products by speed and space considerations. Technically, FORTH is a "threaded" language, but that doesn't imply anything about its implementation. Besides, I love to bug the FORTH aficionados. Anyway, to proceed.

The assemblers grow more numerous almost monthly, and it is obvious that most serious graphics work for the Atari is still being done in assembly language, even though the 6502 has one of the strangest assembly languages in existence. (There used to be others far stranger, but they've either died out or been relegated to the dedicated controller market. You know you're an old-timer if you ever used a 4004, PPS-4, PPS-8, 8008, F-8, 2650, COPS, TI1000, etc.)

Of course, Macro Assemblers are a step in the right direction, but I have yet to see any 6502 assembler system done "right," with relocatable and linkable object modules, a symbolic debugger, and more. Yet. For those of you not familiar with macro-assembly techniques, I should point out that old macro hackers usually build up a library of their favorite macros and can easily plug together several variations on a utility program (for example) by simply picking and choosing from their assortment of macros.

I don't really want to explore this subject in depth right now, but I would like to point out that, using some – or at least one – of the currently available macro assemblers for the Atari, you can write assembly language programs that look like this:

```

OPEN 1,8,0,"D:NEWFILE"
LOOP
  INPUT 1,LINE
  IFERROR EXIT
  PRINT 0,LINE
  GOTO Loop
;
EXIT
...
```

It would seem to me that the percentage of Atari owners who will successfully dive into assembly language is too small to make *any* assembler become the dominant Atari language. Currently, though, there is no other way to write such marvels as Eastern Front, Frogger, and operating systems. So, at least for many software heavyweights, assembly is *the* language.

Compiling 6502 Code

I'd like to skip the interpreters for now and discuss both kinds of compilers. For starters, what's the difference between a compiler and a pseudo-compiler? Software purists could argue this point for days, but I will use a simple rule here: if it produces output, it's a compiler. If it produces tokens or words which must be interpreted, it's a

pseudo-compiler.

Now, quite honestly, on a 6502 there probably isn't much advantage in one of these over the other. Generally, a pseudo-compiler produces fewer bytes of code, but requires a relatively massive runtime support module (the interpreter, including I/O routines, etc.). As a rule, on most computers, pseudo-compiled code will run slower than compiled code because of the overhead of the interpreter.

Unfortunately, most conventional language compilers for 6502-based machines will of necessity produce large and generally clumsy code. Consider the following statement, legal, with minor variation, in most higher level languages:

```
array(index) = value ;
```

Given that all three variables shown are 16-bit globals, a *really* good compiler for a Z80 could produce as few as 15 bytes of code to execute it (and the one we wrote for Cromemco produces only 16 bytes).

A *superb* compiler for the 6502 could produce as few as 25 bytes, but only if it *knew* that "index" would not contain a value exceeding 127! And, oh yes, most pseudo-code compilers would probably produce 11 or 12 bytes of tokens for this same code.

So, you see, even a multi-pass optimizing compiler can *at best* coax the 6502 into using 1.5 to 2.5 times the amount of code that a Z80 needs. And, in truth, there aren't any "superb," "multi-pass," "optimizing" compilers yet available for the Atari. So the code generated will be even bigger, perhaps as much as three to four times that needed by a Z80. (To be fair, an "average" Z80 compiler would produce 25 or so bytes of code, itself.)

So why did we digress through all of this? Simply to show that it is remarkable that there are any compilers at all for the Atari. Of the two compilers shown, the PASCAL is the more complete language, but it is a little difficult to work with, needs a huge support library, and requires two disk drives. Still, since it is an APX product, it is a remarkable bargain. C/65, on the other hand, is a subset of the full C language; it is a one-pass compiler (no optimizing here, obviously) which produces macro assembly language output. Its primary advantages: the assembly language can produce a listing with the original C code interspersed as comments, it uses a very small support library, and it can run on a single drive. But I think we may not have seen the end of compiler efforts on the 6502.

Interpreter Efficiency

But now we come to my favorite topic: interpreters. Despite its shortcomings as a compiled-for machine, the 6502 comports itself nicely when interpreting:

it is fast and needs only relatively compact code to implement. Why? Simply because interpreters generally work on "lines" of input. But if we limit a line to 256 characters (a very reasonable limitation), we find that there are several modes of operation on the 6502 that just love working with such short character strings. (Especially, of course, the "indirect indexed" or "(zero page),Y" instructions.) The truth of the matter is that the designer of a 6502-based interpreter has a lot of leeway in prescribing how the language will run best.

So look at the wealth of interpreters available already! With more to come, I am sure. We find in these interpreters the most used of all Atari languages, Atari BASIC. Well, that's not surprising, considering that it's essentially a required ingredient in an Atari system. But let's come back to it in a moment.

Naturally, PILOT is here. It's a nice, simple language which can easily be interpreted. It was probably a joy to program; I would have loved being involved.

But there are some real powerhouse languages here, also. LISP has traditionally been an interpreter, the darling of the Artificial Intelligence people. And, finally, there is Microsoft BASIC and BASIC A+. Quite honestly, I feel that these last two languages provide the best and easiest access to the Atari's features. Naturally, I am prejudiced towards BASIC A+, but the Microsoft BASIC has a few nice and unique features even if it isn't quite as easy to use.

What's The Atari Language?

So, after all that, just what is *the* Atari language? Well, I'm going to cop out and say that it's Atari BASIC. Despite all the nasty things said about the poor thing, look at all the things written in BASIC. And they work.

Atari BASIC is an excellent starting point. The easiest next step is BASIC A+, but most people won't have too much trouble learning other algebraic languages, such as PASCAL or C (the only real problem with these languages is that debugging is so much harder than with an interpreter). I consider PILOT and LISP useful languages in their own right, but much of what you learn in them is non-transportable to other languages.

The same is true of FORTH. FORTH enthusiasts would have you believe that FORTH is the only language you will ever need. Nonsense. Each language has its uses, its strong points, and its failings. (In my opinion, the major failings of FORTH are (1) that it operates independent of the host system's DOS and (2) techniques learned in FORTH are often non-transportable to other languages, because of FORTH's reverse-Polish

notation. However, I respect the language for what it is: a hacker's dream come true. And I'm a hacker.)

Personally, I like to collect languages the way other people collect games. Seldom will I find one that won't teach me something new about how computers can be made to work. So try some "foreign" languages yourself soon and see how much fun they can be. (And pain and trouble and frustrating and educational and uplifting.)

System Reset And The 850

A couple of times in the past, I have presented in this column the "rules" for adding device drivers to Atari OS. Well, would you believe it, Atari itself broke the rules when they implemented the 850 (RS-232) handlers. The violation was a minor one, yet the consequences *can* be severe. To start with, let's recap my rules:

1. Locate the current value of system LOMEM (contents of \$02E7).
2. Load your driver into memory and relocate it to LOMEM.
3. Adjust the contents of LOMEM to reflect the memory being used by your driver.
4. Add your device's name and handler address to the handler table (HATABS, at \$031A).
5. Get the current value of DOSINI (location \$000C) and save it somewhere in your handler. Put your own initialization address into DOSINI.
6. Whenever your initialization routine is called (i.e., when System Reset is pushed by a user), first call the initializer whose address was in DOSINI before you changed it. Then perform steps 3 and 4 again, since Reset will have changed LOMEM and reloaded the HATABS.

Now step 2 is the most difficult of these to accomplish, in practice, because it is hard to produce a relocatable module on the Atari. Many programs I have seen (and written) are actually assembled absolute at a "known" good location. This is okay, if you are writing for your own private system: you know what will be loaded when and where. But if you are producing a driver for sale, you really should follow the rule faithfully.

Atari's 850 drivers do, indeed, relocate themselves beautifully. They add their name to the handler table. They adjust the system LOMEM pointer. So what do they do wrong? One minor thing: they do steps 3 and 4 *before* they call the old initialization routine (see step 6) instead of *after*!

The result: the 850 handler changes LOMEM to just above itself and then calls the DOS initialization, which resets LOMEM to just above DOS!

Thus, the RS-232 handlers are not protected from programs which come in and *quite properly* use RAM starting at LOMEM. Generally, if you are running with Atari BASIC, this won't affect you, since BASIC maintains its own pointer to LOMEM once it is initialized at power on. But if you return to DOS without MEM.SAV, or run some assembly language utility... well, there are just too many cases where this little *faux pas* can wipe you out.

I am currently working on a patch (ready by next month, I hope) to the handler (to be made via the handler loader) which will fix this problem. In the meantime, it might be a good idea to have your programs check for the existence of the "R" name in HATABS and avoid the appropriate amount of memory if it is found.

In December we'll have some heavy assembly language stuff, what with the patch to the 850 handler and the cassette verify routine. I hope to return to some more BASIC stuff to start off the new year.

COMPUTE!

The Resource.

CAPUTE!

Modifications Or Corrections To Previous Articles

TI 99/4A Charades

Our thanks to Steve Davis, author of "Charades" for the TI 99-4A (September 1982, p. 64), for pointing out the following typos in the program listing:

```
831 DATA SHARP AS A TACK
1220 CALL CLEAR
1330 IF STATUS=0 THEN 1340 ELSE 1350
1370 RETURN
1580 CALL SOUND
```

PET Machine Language Compactor

Author David Evans has provided some readers with a faster version of his "Compactor" (July 1982, p. 159). To make all versions work correctly, he suggests that the following line be typed in and then the corrected version be saved via the monitor (the start and end addresses are \$0363, \$0B78):

```
IF PEEK (2461) = 12 THEN POKE 2461,13
```

NEW FOR ATARI

FROM

MMG MICRO SOFTWARE

*****NECESSITIES*****

DISK COMMANDER - Just save this program on your BASIC disks and it will autoboot and automatically list all programs from the disk into your screen. Simply run any program by typing in a number.

Requires 16K, Disk Only \$24.95

BASIC COMMANDER - This all machine language program is an absolute requirement for ATARI BASIC programmers. Single keystroke DOS and BASIC commands, plus: AUTONUMBER, RENUMBER, BLOCKDELETE and much more!

Requires 16K, Disk Only \$34.95

RAM TEST - The most thorough and fastest memory test available for the ATARI. This all machine language program takes 4 min. to test 48K. It's the only program that tests the cartridge area of RAM. Good for new 400/800 computer owners and for testing new RAM boards.

Requires 8K, Disk or Cassette \$24.95

*****BUSINESS/HOME*****

MAILING LIST - Extremely fast BASIC and machine language program. Each data disk holds over 500 files. Sort on any of 6 fields at machine language speed. Use any size labels or envelopes.

Requires 48K, Disk Only \$39.95

*****TUTORIALS*****

ASTEROID MINERS - A unique game tutorial. A 32K BASIC game utilizing over 25 players in player-missile graphics, machine language subroutines, a redefined character set, multiprocessing utilizing the vertical blank interrupt interval, and much more! Comes with a book documenting each part of the entire program, and fully documented source code for both the BASIC and assembly language parts of the program. Use these routines in your own programs. These examples will make it easy.

Requires 32K, 1 Joystick - Cassette or Disk \$34.95

Dealers and Distributors Contact:

CLASSIC SOFTWARE, INC.

RD1 - 3D • HIGHWAY 34 • MATAWAN, NJ 07747

(201) 566-5007

ATARI is a registered trademark of ATARI, INC. N.J. Residents add 5% sales tax



ALL MACHINE LANGUAGE ARCADE GAME WITH INTELLIGENT MONSTERS!

This arcade style game is sure to become an ATARI classic. Chomper requires 16K RAM, 1 joystick and nerves of steel.

Available on Disk or Cassette \$29.95

Available at your favorite computer store or
Send a check or money order directly to:

MMG MICRO SOFTWARE

P.O. BOX 131 • MARLBORO, NJ 07746

or call (201) 431-3472

for MasterCard, Visa or COD deliveries

● Insight: Atari

Bill Wilkinson
Optimized Systems Software
Cupertino, CA

This month, I will follow through with at least one of my promises for some heavier assembly language stuff: the discussion and source for the fix to the 850 handler LOMEM problem. Unfortunately, I did not manage to complete the other promised project, the BASIC Cassette Verify program.

That program has proven more difficult to write than I had suspected it would, primarily because it's hard to get the debugger and BASIC to cooperate. With some luck I will have the problem fixed very shortly.

In any case, I've also got a few little tidbits to share with you, so let's tackle them first.

Atari-CP/M Revisited

First, I would like to clear up a misunderstanding (on my part) about the Vincent Cate (USS Enterprises) Atari-to-CP/M connection, mentioned a couple of issues ago. I stated that one problem with the system was that you would not be able to use standard Atari diskettes. Not totally true. If you have (or have access to) an Atari compatible 810 drive, you *can* copy programs from the 810 to the CP/M host. (Vincent claims that the system is even capable of properly simulating self-booting disk games, etc., though I would imagine that some of the heftier protection schemes might defy his standard system.)

Anyway, the address for USS Enterprises is 6708 Landerwood Lane, San Jose, CA 95120. I hope this doesn't seem too much like an ad or endorsement: I have *not* used the system. I have, however, heard from people who have and who say it does what it claims to do.

In the same column, I mentioned a new product to be introduced soon which would function either as an Atari disk controller (810 emulator) and/or as a CP/M system in which the Atari console was a smart terminal. That project is apparently at the reality stage, so I guess in fairness I should now mention it by name.

The company producing the product is Software Publishers, Inc., of Arlington, Texas. (I know,

I know. *Software* publishers?) The base price of the controller, I have been told, is about \$500 without disk drive. The CP/M add-on will be (is?) about \$250. Perhaps someone will soon give us a review of the viability of this concept.

Double No-Trouble

Speaking of viability: We have been using our Percom drives (one double density, one double sided and double density) for about three months now. We are more than satisfied with their reliability. And, of course, the new OS/A+ we produced for use on the larger drives allows considerable flexibility. Perhaps the Atari can be used as a business machine after all.

And to be sure that we don't slight anyone, I need to mention that our MPC double density system has been here about a month now also and seems to be working fine.

So far, all the things we've tried seem better for most purposes than the 810 drives, though all of them seem to have trouble with some heavily protected diskettes. Moral: buy the drive, forget the diskettes. (Side issue and pet peeve: If it's that heavily protected, it will have trouble even on a slightly out of speed Atari 810. So far, I have plunked down my scarce dollar only three times for copy-protected disks. I think I will try to be thriftier in the future.)

Percom DOS

By now it should be general knowledge that the "new and improved DOS" that Percom has been publicizing is none other than OS/A+. But it is a significant change from our "old" OS/A+, which is really just a CP/M-like keyboard interface hooked to the Atari DOS 2.0S File manager. Thanks to the efforts of Mark Rose, our youngest associate and a junior at Stanford University, we have managed to produce an all new, random access DOS designed to interface to any and all disk drives from 128 kilobytes to 16 megabytes. The "random access" description implies that you are not tied to the tyranny of NOTE any more (and POINT is now reasonable: you POINT to a byte position *within a file*, just like on the big guys' systems, and better than CP/M).

This may sound like an advertisement for OSS and Percom, but it really isn't. First of all, our profits aren't really tied to the sales of this new DOS, so it isn't really an ad for us. And second, it appears that OS/A+ will be used by all the other Atari-compatible drive manufacturers, so Percom is offering it first but not alone. Anyway, the real reason I brought this up (aside from wanting to pat Mark Rose on the back in public) is to pass on a few of the things that you should watch out for if you are thinking of moving to either more or larger

drives.

LOMEM On The Tot-Mem Poll

I am sadly dismayed to see so many Atari-produced and Atari-compatible products being introduced nowadays which violate one of the prime rules for running on an Atari: *don't put anything lower in memory than LOMEM*.

After all, the operating system provides these nice, convenient locations LOMEM and HIMEM, which contain the addresses of the bottom and top of usable memory. Why not use them?

But no, let us assume that we will run under Atari DOS 2.0S, with two single density drives, with our blinders on (so that we cannot see the future). Phooey. How about a little table to show the values of LOMEM under various DOS configurations, with various numbers of drives and files available?

LOMEM With Various DOS's

Dos Used	Number Of Drives	Number Of Files	Contents Of LOMEM
Atari DOS 2.0S	2-S	3	\$1C00
Atari DOS 2.0S	4-S	7	\$1F00
Atari DOS 2.0S	2-D	3	\$1E80
Atari DOS 2.0S	2-S, 2-D	5	\$2180
Atari DOS 2.0S	4-D	7	\$2380
OS/A + ver 2.0	2-S	3	\$1F00
OS/A + ver 2.0	4-S	7	\$2100
OS/A + ver 2.0	4-D	7	\$2680
OS/A + ver 4.0	2-D	3	\$2C00
OS/A + ver 4.0	4-DD	7	\$3300

legend: -S means single density drives
 -D means double density drives
 -DD means double sided, double density

Surprised? It gets worse: if you load the RS-232 handler for the 850 Interface Module, you must add almost \$700 to all the table figures! (And I left out K-DOS simply because I don't know the correct figures there, but I understand that they are all over \$3000.)

"But," you say, "how come you show Atari DOS with double density drives?" Aha! You didn't know that Atari DOS will handle double density drives for most user programs? (The menu can get confused, especially for duplicating disks, but BASIC - for example - runs just fine.)

We agonized a long time over coming out with OS/A + version 4, the Percom (et al.) random access DOS, with its much higher LOMEM values. But then we realized that, given that you will use double density and larger disks, there is simply no way to stay completely compatible. So, if you're going to do it, do it right.

Incidentally, Percom's initial patches to Atari DOS 2.0S solved the problem in a different way: they moved the disk buffers to the top of memory

and dropped HIMEM. Of course, then they ran into trouble with the programs that ignore HIMEM. Like BASIC A+? Welllllll, I guess we have to take our lumps, too. Sigh. But we're working on it, honest.

So this has gone on long enough. The moral: if you're writing assembly language programs, pay attention to the rules. If you're stuck with an interpreter or compiler that does it wrong, go yell at the company that palmed it off on you.

Mishandler

Since I am ranting on about LOMEM anyway, let's tackle the problem I presented last month: the Atari RS-232 handler for the 850 Interface Module does not handle the RESET key properly when the disk device (or other previously loaded handlers) is present.

The result is that LOMEM will be reset to what the disk handler thinks it is, rather than above the 850's driver. And, of course, this means that any program which uses LOMEM properly will zap the RS-232 (Rn:) drivers. Which might not be so bad except that the Rn: name will still be recognized by CIO. Which might be a real disaster.

Why did all this come about? Because Atari didn't follow their own advice. When you steal DOSINI from DOS, in order to link yourself into the RESET chain, the *first* thing you should do is call the old DOSINI. Instead, the 850 handler does all its initializing, resets LOMEM to above itself, and *then* calls the old DOSINI! (And, of course, poor old FMS doesn't know that R: exists, so it moves LOMEM to just above itself. And, admittedly, you *could* fix the problem by having DOS change LOMEM only if the change is upward. This is left as an exercise to the reader.)

So what do we do about this bug? If you are using BASIC (or BASIC A+), forget about it. BASIC maintains its own LOMEM pointer, which is initialized only at BASIC coldstart time (e.g., at power-up). In fact, many system programs either do similar things or have been purposely assembled in higher memory to avoid all possible drivers. (Except see that good old table. Maybe they aren't all high enough?)

However, if you need to fix this problem, chances are you need to fix it quickly and thoroughly. The machine language program below seems to do a reasonably good job of patching the mess. But, of course:

Caveats: (1) This program works as shown with my 850 Interface Module. I know for a fact that Atari has made more than one version of this beast, so I can *not* guarantee it will work on yours. (2) This program works by patching the AUTO-RUN.SYS (also known as AUTORUN.232 or

RS232.OBJ or RS232.COM) file. If you are not using Atari DOS (or OS/A+, for RS232.OBJ or RS232.COM), then this will work only if you can load and execute this routine at the addresses shown in the listing.

So how does this program work? To understand it, we must first understand how the Rn: handler is loaded from the 850.

Here I Am

When the Atari computer is powered up, it finds out if a disk drive is attached by sending out a status request command (via SIO). If, indeed, disk drive number one is alive and well, then the disk boot proceeds. But if the 850 is alive and well, it is also sitting on the serial bus, looking at SIO sending status request command(s) to the disk. SIO will try 13 times to boot the disk before giving up. But here is where the 850 gets sneaky: if the disk doesn't answer after about ten of those tries, the 850 jumps on the bus and says "Here I am! I'm the disk drive! Boot me!"

And, of course, the computer indeed "boots" the disk — whether it actually is the drive's controller chip responding or whether it is an 850 in chip's clothing. And that's how those 1800 or so bytes of code get into the computer when all you have is an 850.

But how does that code get pseudo-booted when you *do* have a disk? Well, one way would have been to distribute the handler on the disk. But why waste all that good code sitting out in the 850, just waiting to be executed? So AUTORUN.SYS (in any of its aliases) is a very small routine that performs just the right operations to load the 850's serial handlers.

In building the program presented here, I have cheated. Quite frankly, I have not investigated why and how the code used in AUTORUN.SYS works. And quite frankly, I don't care. What I have done is simply build my program around that code. And here's what my program does.

First, I get the current contents of DOSINI (presumably the address of the FMS initialization routine) and save them for later use. Then I fall through and let the 850's code be loaded and initialized. If this process is successful, I then find the new contents of DOSINI (the Rn: driver's initialization routine address) and save them also. And where do I save the two initialization addresses? In the middle of the patch to be applied to the 850 driver.

Then all I need do is move the patch into the middle of the driver and relink DOSINI to point to the patch. Now, the cute part of all this is: where do we put the patch? Why, right on top of the erroneous call to the FMS initialization. (The one

that occurs *after* the 850 init, remember?)

Ummm, but I'm patching a JSR to the FMS init followed by a JMP to the 850 init. How does all that fit into the space of one (previous) JSR? And what about the code immediately preceding the patch? Here it comes, the kludge. The code we are replacing includes a check of the warmstart location, since the handler does not bother to call the FMS initialization if it doesn't need to. Well, with our code patch, the FMS always gets called to init itself. But so what? It doesn't hurt anything, just slows the loading of this 850 interface code an unnoticeable amount.

Anyway, if you can follow the code, you will note where the patch is being applied. The byte immediately before the patch location *must* be a CLC instruction. (Check it out by loading the RS-232 handlers and then using a debugger to list the code.) If it is not, then your 850 differs too much from mine to use this routine as is. (And if you figure out where to patch it, why not tell all of us.)

Last but not least, notice that the patch is intrinsically relocatable, just as is the 850 handler. It should work in virtually any memory and/or disk drive and/or DOS configuration.

Whew! That was lengthy and heavy, right? Well, cheer up, there's more to come next month. Like how to add a default drive specifier to Atari DOS and OS/A+. If you have two drives, wouldn't it be convenient to be able to specify that "D:..." meant "D2:..." once in a while? Watch this space.

Atari 850 Fixer Upper or: when in doubt, punt.

```

0000      1010      .PAGE " or: when in doubt, punt."
          1020 ;
          1030 ; Some equates
          1040 ;
0043      1050 FIXOFFSET = $43      ; read the text
000C      1060 DOSINI = $0C        ; the cause of all this
          1070 ;
          1080 ;
          1090 ; This first code is simply to save the original
          1100 ; contents of DOSINI for later use, like the
          1110 ; 850 code should have done in the first
          1120 ; place. Sigh.
          1130 ;
0000      1140      *= $3800-10
          1150 NEWLOADER
37F6 A50C  1160      LDA DOSINI      ; presumably, we are saving
37F8 8D7738 1170      STA PATCH2+1  ; the FMS init vector for
37FB A50D  1180      LDA DOSINI+1    ; later use, but the beauty of
37FD 8D7838 1190      STA PATCH2+2  ; this: it works w/o FMS also
          1200 ;
          1210 ;
          1220 ; Now we begin the original Atari loader code.
          1230 ;
          1240 ; If your code doesn't agree with this, it
          1250 ; is possible that your 850's internal
          1260 ; is different also. If so, apply the
          1270 ; patches with caution. Read the text.
          1280 ;
          1290 ; CAUTION: this code is uncommented, simply
          1300 ; because I'm not sure exactly what it
          1310 ; is doing. But who cares...it works.
          1320 ;
3800      1330      *= $3800      ; where the Atari code was found
          1340 LOADER
3800 A950  1350      LDA #$50
3802 8D0003 1360      STA $0300
3805 A901  1370      LDA #$01
3807 8D0103 1380      STA $0301

```



```

380A A93F 1390 LDA #S3F
380C 8D0203 1400 STA $0302
380F A940 1410 LDA #S40
3811 8D0303 1420 STA $0303
3814 A905 1430 LDA #S05
3816 8D0603 1440 STA $0306
3819 8D0503 1450 STA $0305
381C A900 1460 LDA #S00
381E 8D0403 1470 STA $0304
3821 8D0903 1480 STA $0309
3824 8D0A03 1490 STA $030A
3827 8D0B03 1500 STA $030B
382A A90C 1510 LDA #S0C
382C 8D0803 1520 STA $0308
382F 2059E4 1530 JSR SE459
3832 1001 1540 BPL $3835
3834 60 1550 RTS
3835 A20B 1560 LDX #S0B
3837 BD0005 1570 LDA $0500,X
383A 9D0003 1580 STA $0300,X
383D CA 1590 DEX
383E 10F7 1600 BPL $3837
3840 2059E4 1610 JSR SE459
3843 3006 1620 BMI $384B
3845 200605 1630 JSR $0506
3848 4C4C38 1640 JMP FIXIT ; this WAS a 'JMP (DOSINI)'
384B 60 1650 RTS
1660 ;
1670 ; Now the 850 has loaded its code into memory...
1680 ; so we can patch its boo-boos
1690 ;
1700 ;
1710 ;
1720 FIXIT
384C A50C 1730 LDA DOSINI ; The 850 code has patched
384E 8D7A38 1740 STA PATCH3+1 ; its init entry point into
3851 A50D 1750 LDA DOSINI+1 ; 'DOSINI' ... we will jump
3853 8D7B38 1760 STA PATCH3+2 ; to it at the end of our patch
1770 ;
3856 A043 1780 LDY #FIXOFFSET ; for my 850! read the text
3858 A200 1790 LDX #0 ; loop index
1800 ;
1810 ; We move our patch code into the 850's code
1820 ;
1830 PATCHLP
385A BD7538 1840 LDA PATCH1,X ; a byte of patch...
385D 910C 1850 STA (DOSINI),Y ; into the 850 code
385F C8 1860 INY ; next patchloc
3860 E8 1870 INX ; next byte of patch
3861 E008 1880 CPX #8 ; unless done
3863 D0F5 1890 BNE PATCHLP
1900 ;
3865 A944 1910 LDA #FIXOFFSET+1 ; again, caution...read text
3867 18 1920 CLC
3868 650C 1930 ADC DOSINI ; we move DOSINI to point
386A 850C 1940 STA DOSINI ; to our patch...which in
386C A50D 1950 LDA DOSINI+1 ; turn will jump back to
386E 6900 1960 ADC #0 ; the 850's init code.
3870 850D 1970 STA DOSINI+1
1980 ;
3872 6C0C00 1990 JMP (DOSINI) ; and this actually goes to our
; patch!
2000 ;
2010 ;
2020 ; This patch area has two addresses placed
2030 ; in it and then it is moved en masse
2040 ; into the 850 code, as a patch thereto
2050 ;
2060 PATCH1
3875 60 2070 RTS ; gets rid of some unneeded code
2080 PATCH2
3876 200000 2090 JSR 0 ; becomes JSR FMSINIT, or some
; such
2100 PATCH3
3879 4C0000 2110 JMP 0 ; to original reset point
387C 00 2120 BRK
2130 ;
2140 ; This is just to make it a LOAD AND GO file
2150 ;
2160 ; You might wish to use $2E2 instead if you
2170 ; understand the implications thereof
2180 ;
387D 2190 * = $2E0
02E0 F637 2200 .WORD NEWLOADER
02E2 2210 .END

```

* **Commodore**

*VIC 20 \$176
16K RAM \$ 79



MEMORY EXPANSION

VIC 1540 DISC DRIVE \$299.95
VIC 1530 DATASSETTE \$ 64.95
VIC 1515 GRAPHIC PRINTER \$299.00
VIC 1213 MACHINE LANGUAGE MONITOR \$ 43.00
4 SLOT FULLY BUFFERED EXTENDER \$ 59.95
VIC MODEM \$ 89.95
8K RAM MEMORY EXPANSION \$ 39.95
2Kx8 STATIC RAM CHIPS (200 NSEC) QTY. _____ ea. \$ 7.95

ORDER FORM

(Circle Above Items)



NAME _____
STREET _____
CITY _____
STATE _____ ZIP _____
PHONE _____

CHECK ONE:



☐ VISA ☐ MASTERCARD
☐ Check Enclosed ☐ C.O.D.
Credit Card # _____
Expiration Date _____

Add 3% Shipping Charge.
COD's add \$1.50 plus 20% Deposit
Required. CA Res. 6% Tax
Credit Cards add 3%
Personal checks accepted
(Allow 3 weeks extra)

U.S. TECHNOLOGIES

P.O. Box 7735
San Diego, CA 92107
(619) 224-8016

WE WILL NOT BE UNDERSOLD!

DEALERS: REDUCED
PRICES OFFERED ON
LARGER ORDERS
CALL FOR DETAILS

*Trademark of Commodore

COMMODORE ★ PET OWNERS ★ NEW AUTHENTIC PROGRAMS

CASINO CRAPS

- Any bet made in Vegas, now can be made at home.
 - The Field Hardways- Place Bets-Come-Pass Line
 - Find a winning system, without losing a dime.
- 8K version (1 player) \$10.95
16K version (5 players) \$12.95

KONNECT FOUR

- Now play this popular game against your pet.
 - Excellent sound & graphics
 - Real time clock
 - Three levels of play
 - Can fit into 8K
 - Fun & Educational for all ages
- ONLY \$10.95

GPMicrosystems
72-31 67th Place
Glendale, N.Y. 11385

Please include \$1.50 shipping
& handling for each program.
Indicate version.

INSIGHT: Atari

Bill Wilkinson

No gossip to start with this month. Instead, let's start right off into a whole series of interesting tidbits (and even a few tidbytes).

Which Is It? GTIA Or CTIA?

Several articles have been written on how to tell whether you have a GTIA or CTIA in your system. Most of them suggest that you use a GRAPHICS 9 statement and observe the screen (it turns black with a GTIA, remains blue with a CTIA).

But suppose you want to write a program that takes advantage of all the capabilities of the GTIA. What does the poor user with only a CTIA do? If you are commercially clever, you will have your program sense which chip is in use and adapt itself accordingly. This program will enable you to do just that:

```
100 GRAPHICS 0:REM ALWAYS USE THIS MODE
110 PRINT "NOW TESTING FOR CTIA VERSUS GTIA"
120 PRINT "=====
30 POKE 559,58:POKE 53277,2:REM ENABLE PLAYERS
40 POKE 54279,240:REM USE ROM FOR PLAYER DATA
145 POKE 53248,80:REM CENTERED PLAYER
150 POKE 53278,0:REM CLEAR COLLISION REGISTERS
160 POKE 623,65:REM ENABLE GTIA, IF IT EXISTS
170 POKE 20,0
180 IF PEEK(20)<2 THEN 180 Change this!
190 POKE 623,1:REM DISABLE GTIA
200 POKE 559,34:POKE 53277,0:REM TURN OFF
PLAYERS
210 FOR A=53261 TO 53265:POKE A,0:NEXT A:REM
(AND PLAYER DATA)
220 IF PEEK(53252) THEN PRINT "SORRY, ONLY A
CTIA":GOTO 240
230 PRINT "AHA! A GTIA."
240 END
```

First of all, to give credit where it is due, I should mention that I was inspired to try this by a remark I read in one of Craig Chamberlain's articles in the M.A.C.E. newsletter (Michigan Atari Computer Enthusiasts, in the Detroit area). Portions of that article were also reprinted recently in **COMPUTE!**.

But let's now discuss the program. First, we must explain why and how it works: there is no way to inquire of the chip itself which it is. Even the operating system does not know which is installed. But (There *had* to be a "but," or there wouldn't be this article.)

There are a few subtle differences between how the two chips view players and missiles. In particular, the GTIA doesn't believe that players can collide with "printed" characters, so it never reports such a collision. The CTIA, though, con-

siders a character to be just another kind of COLOred (and SETCOLOred) display.

The first thing our listing does is insure that we have some mode zero characters on the screen (lines 100 - 120). Then we enable the player DMA and the players themselves (line 130). And we tell the chips that the player data memory is smack in the middle of the ROMs! (Why? To insure that lots of data bits will be on, forcing lots of collisions between the player and the playfield screen characters.)

With line 145, we place the player somewhere left of the center of the screen, insuring that it will collide with our printed message. Then, after clearing the collision registers (to insure that the later results will be valid), we enable the special modes of the GTIA (lines 150 and 160).

We wait for at least one full screen scan (lines 170 and 180), to be sure that the collision will "take" (if it's going to). Then we turn everything back off again (lines 190 - 210).

Finally, we inspect the collision register for player zero. If a collision did occur, it must be because the older CTIA was installed. If no collision occurred, we presume that we have a GTIA.

All of this is a little complicated, but I sincerely hope that some of you game developers out there will start designing some good GTIA-based games, now that you can have them modify themselves for the CTIA owner.

A Few Abbrev'd REMs. Period.

In his article on "The Atari Wedge" (in the November 1982 **COMPUTE!**), Charles Brannon mentions that BASIC treats a line beginning with a period as a REMark, claiming that it is a lucky fluke. Well, it really isn't a fluke. It's just one of those things that got designed into Atari BASIC and then forgotten about.

The rule for using abbreviations in Atari BASIC (and BASIC A+, naturally) is fairly simple: when a statement begins with an abbreviation (any alphabetic characters followed by a period), BASIC searches the keyword name table for the first statement name which matches the abbreviation, starting at the first character of the abbreviation and ending at the period.

This means, for example, that "L." will match "LIST" only because LIST is the first word in the

keyword name table that begins with an "L". If "LET" had been placed before "LIST" in this table, then "L." would have been interpreted as a LET statement. Boy, aren't we lucky that LIST comes before LET!

Luck had nothing to do with it. The order of those keywords was carefully chosen to provide the maximum usability of the shortest abbreviations. (Actually, I now believe that there are a few variations in the order that might be more useful; but remember that the order was set by intuition, not experience, since the language didn't then actually exist.)

Anyway, Atari had asked for a very short abbreviation for REMark statements (e.g., "!", as is used by most Microsoft BASICs). But what could be shorter than a single period? It's even easier to use than "!" (no shift key needed). How to produce that result? Trivial! Place REM as the first statement name in the keyword table.

So try it sometime. Why type in three characters ("REM") when one will do? Of course, because of the tokenizing nature of Atari BASIC, any abbreviated statement(s) are LISTed in their full form. So "." will be LISTed as "REM".

And a P.S. for those of you into BASIC internals: note that this implies that the token value for REM must be zero, since the token values relate directly to the order of the names in the keyword table.

Page 6 Preached Again

I kind of promised myself that I would get down off my soap box this month and quit ranting and raving. But I couldn't go one whole column without a little preaching, could I?

Stay out of page 6! I can't believe it! It seems that every other article and/or utility program and/or device driver that I run across wants to place itself in page 6 (memory locations \$600 to \$6FF, 1536 to 1791 decimal). *It won't work!*

How can I possibly install a printer driver in page 6 and then put my player vertical move routine there and my disk block input and output and Ah, come on, folks. Give us a break.

If you are writing a complete "system" (a game, or data base program, or whatever), then you are naturally free to configure memory as you wish, including doing whatever you want to page 6. But if you are going to publish a utility in a magazine or include a device driver with your printer interface board or do anything that others might use or modify, *please* don't make it fixed-assembled in page 6. *Please.*

Besides, it is *not* true that BASIC leaves all of page 6 alone. If you do an INPUT from disk (or cassette or anything other than the screen), and if the data you input exceeds 128 bytes, BASIC will use at least a portion of page 6 as its buffer. (How-

ever, it is probably – not surely, just probably – safe to use memory from \$680 to \$6FF.)

A little history: If you examine your Atari BASIC reference, you will find that there are two memory usage tables. One claims that all of page 6 is available for the user. The other claims that only the upper half is available. In general, you should believe the latter. *It is not a design flaw nor an error* that BASIC sometimes uses the bottom half of page 6. It is necessary and documented.

I think it was someone at Atari (my rumor sources say Chris Crawford, but this is unconfirmed) who began using all of page 6 for assembly language routines. And, as I stated above, there is really nothing wrong with doing so within a "closed" environment (where you write *all* the software, both BASIC and assembler). Just don't do it for public consumption.

So what should you do, instead? The best solution is to write self-relocatable code and load it wherever there is free memory (e.g., in a BASIC string). (Showing how to write self-relocatable code might be an instructive article, in and of itself. Any takers?)

The second best solution is to perform my favorite trick: place your code at LOMEM and move LOMEM up. Even here, though, it is best to use relocatable code, so you can run under a variety of operating system configurations and varying heights of LOMEM (as I documented in last month's column).

And, last but not least, I have some good, practical (and a little bit selfish) reasons for avoiding page 6: BASIC A+ uses a good portion of it (\$610 through \$642, actually). Does that make us a villain? Perhaps a little, to the article writers. But we aren't that terrible: I understand that Microsoft BASIC uses *all* of page 6. And who knows what other languages and operating systems and peripheral devices and whatever will also use page 6? Why complicate both your and others' lives by putting your routines there also?

Some FORTH-Right Comments

I received a very well written and thought-out letter from Steven Weston, of Del Mar, California, regarding the benchmarks I reported in my September 1982 column. Mr. Weston shares the predilections of some others, considering FORTH to have been slighted in that column (and in the following one, I presume).

First, I should like to report that he translated the BASIC benchmark to FORTH and obtained a time of a little under 118 seconds. Which is interesting, since ValFORTH (the version he used) makes use of the Atari floating point routines, I believe. So why should it be slower than Atari BASIC? If I were guessing (which means I'm about

to take a flyer), I would presume that the floating point words for ValFORTH are written in FORTH words, instead of being written as low-level (assembly language) words. The very operation of stacking and unstacking the floating point numbers must then be relatively slow and painstaking.

If this is indeed true, then my comment is a positive one: the FORTH user indeed has the choice of implementing "commands" (words) either way, with other FORTH words or with assembly language. This flexibility is poorly supported by most other languages. (Although many C compiler implementations come close to having such accessible assembly code. C/65 functions, for example, need very little overhead in the assembly language code to "unstack" their parameters.) Want a faster FORTH instead of a smaller one? Recode some routines in assembly language.

What Benchmarks Really Test

Before going on to the second point of Mr. Weston's letter, I should like to note that I feel that perhaps he (and many other readers) missed part of the point of the benchmarks: I was really trying to show how useless any one benchmark is, since it is so easy to dream up benchmarks which show off the best features of a given language. I would be hard pressed to construct even a set of ten benchmarks which would adequately compare languages.

And even if I thought I succeeded, how much is the human interface to a language worth? PILOT is still the easiest language on the Atari to learn and interface to. *By definition*, it therefore out-benchmarks every other language for beginners. But would anyone seriously propose using PILOT for generating prime numbers? I think not. Benchmarks are usually worth the paper they are printed on and no more.

So now to Mr. Weston's second point. I quote: "...the bottom line on languages is to use that language which is best suited to the task. [With Atari BASIC] the lack of integer based math is a serious deficiency which can preclude its use by professional software authors." He goes on to ask why I don't provide a "toolbox" of integer math routines to be interfaced to Atari BASIC "instead of defending an inadequate situation."

Well. Kudos and jibes all in one it seems. Anyway, he is absolutely right: pick the language that fits the job instead of making the job fit the language. You will remember, I hope, that in a recent column I mentioned that I collect languages like some people collect games. I keep hoping to find one that will be useful to me.

But now let me disagree a little on a couple of points. And I do so because I have received too

many comments in this same vein. (1) Integer math is *not* needed by all "professional software authors." The person writing a financial package needs integer math about as much as the game writer needs floating point. If you need integer math, choose a language which supports it. (2) BASIC is, unfortunately, a non-extensible language. Sure, we could put integer math routines in memory somewhere and use them from BASIC. But BASIC would still insist on thinking of its variables and constants as floating point, and the conversion time (from floating point to integer to floating point, *ad nauseam*) would wipe out all speed advantages gained. (3) I don't think Atari BASIC is an "inadequate situation." Sure, I think there are other solutions. Why else would our company produce languages such as BASIC A+ and C/65 (and probably more to come)? But "inadequate"? I think not, if it is used for and how it was meant to be used. (If anything is inadequate, it is the 6502 microprocessor, which does not lend itself to the implementation of powerful language compilers.)

But, if you are a beginner, don't let anyone (including me) pressure you into trying to learn a new language before you are ready. It is true that you are not going to write "Super Invading Packers with Tronic Fighters" with Atari BASIC. But just look at what you *can* write! Ten years ago, a computer fanatic would have sacrificed his left thumb for what we now take for granted. Seven short years ago, the "hot" computer game that everybody was rewriting (to make it fit in their expanded memory 8K byte gigantic machine) was Wumpus. Today, I seldom see a published program that doesn't make Wumpus look like something out of the dark ages. Hang in there, folks, you ain't seen nuthin' yet.

The New BASIC Standard?

Well, I finally got time to take a long, hard look at the new ANSI BASIC specification. Whew! I think the tower of Babel must have seemed organized by comparison. Even ADA and PL/1 look like closely designed languages compared to ANSI BASIC. I think that the rule in designing it was "If someone wants it, let's put it in."

You certainly won't see any microcomputer interpreter implementations of it in the near future. I estimate it would take over 80K bytes of Z-80 code to do it (which translates to maybe 100K to 120K of 6502 code). It is definitely designed to be compiled, not interpreted, and then only by big machines.

The error descriptions alone would take a few kilobytes (and they are required!). And what do lines like the following mean?

MAT PRINT #N, SAME, IF THERE EXIT FOR: AS;
B\$, C
ASK #3: access outin, organization org\$, rectype
internal, pointer p\$

I am disappointed. I had hoped that the committee would distill the best of the various BASICs and come up with a somewhat enhanced version of the original ANSI standard BASIC. Instead, they seem to have distilled out the biggest features of the biggest BASICs they can find. And who will use the standard? Not the micros. (At least not in the near future. I understand that Microsoft's representative on the committee dropped out. From frustration? I would have.) Not those who need to contract with the government. (Soon, you will *have* to use ADA if you work with the defense department and various allied agencies.) Not the big business computer users. (They can't afford to go from COBOL, a clumsy but eminently maintainable language, to a BASIC as kludged up by the committee, with a lack of the data structures that made COBOL successful.)

I guess I believed that the only BASIC users that would be left in a few years would be the hobbyists and the time-sharing companies. Now, I think the only ANSI BASIC users will be the time-sharing companies. Maybe.

As much as I disagree with much of what Microsoft has done, I would rather have seen Microsoft BASIC (version 5, on the CP/M machines) become the standard than the hodgepodge the ANSI committee has selected. ANSI, on a scale of 10, I give you a 2.

The New Atari Computers



Perhaps by the time you read this, the new Atari computers will be on display at the Consumer Electronics Show (early January, in Las Vegas). Don't expect any real surprises. I expect to hear of a 64K machine (with no software to take advantage of the extra 16K). And probably a low-end 16K machine.

Obviously, Atari needs to get in there and fight with Commodore, both on price and features. Price is easy. Features? Well, if Commodore follows through as they claim they will, it could be a tough fight. And I think the 400 replacement might outstrip the VIC-20. I guess I should note that I am not as much of an Atari loyalist as this paragraph makes me sound. It's just that I like a good, competitive race. The consumer is bound to win.

Oh, yes, one more thing. No more right-hand cartridge slot in the new machines. And no memory board slots at all. Ouch? I don't know. I hope there will be a good way to expand the new machines, but we will all have to wait to see what it is.

Basically BASIC

All this talk about benchmarks and ANSI BASIC has made me regain interest in a project I thought of doing a while back. So, starting next month, we will begin writing a BASIC interpreter right here in this column. And we will write it in BASIC. Interested? I am.


ATARI

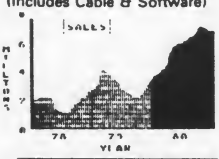
GRAPHICS HARDCOPY

Dumps anything on screen of ATARI 400/800 to printer. All graphics and text modes. Players/Missiles/scaling/grey scale/GTIA/more! Works with EPSON, Centronics 739, IDS and Trendcom.

NEW LOW PRICE
\$79.95
(Includes Cable & Software)

*ATARI is a registered trademark of ATARI Computer Inc.
Specify 400 or 800 and Printer when ordering
(209) 634-8888/667-2888

 **MACROTRONICS, inc.** C.O.D.
1125 N. Golden State Blvd.
Turlock, California 95380



PRINT, READ, & TO PRINT SCREEN DISPLAY



FIRST BORN IN 1978!

the original & continuously updated

CCA Data Management System

Now Available For Atari Computers	\$ 99.50
For Apple Computers	150.00
For CPM Based Computers	225.00

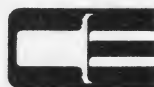
CCA Data Management System

Uses

- Business
- Accounts Receivable
- Accounts Payable
- Inventories
- Billing
- Lists and Rosters
- Home Phone Lists
- Budgets, Hobbies

Features And Capabilities

- Long record lengths
- Up to 24 fields per record
- **Not Copy Guarded**
- Alpha numeric items
- Numeric only items
- Add, update, scan, etc. files
- 10-Level sort ascending, descending, allows alphabetizing data file.
- Contact your local dealer for details or write us for our catalog



DIVISION OF CUSTOM ELECTRONICS, INC.
SOFTWARE

238 Exchange St., Chicopee, Massachusetts 01013
(413) 592-4761

Mastercard & VISA Accepted

- Dealer And Distributor Inquiries Invited
- Closed Mondays — Open Daily Til 5:30 — Fridays Til 8

INSIGHT: Atari

Bill Wilkinson

This month we will examine the possibility of a "default" drive number under DOS. There is also a tidbit about initializing DOS disks from BASIC. Next month, we will begin what will be a three- or four-month series on how to write your own BASIC interpreter.

Deriving The Drive

First, let me state that I do *not* recommend this section to the relative novice. While it is true that you can perform the operations I am about to describe entirely from BASIC, it is also true that you can destroy memory very nicely if you slip up. Enough warning. To begin:

Have you ever (often?) grumbled over the fact that you have to specify not only the file name, but also the disk name and drive number (e.g., "D2:MISSILE.CMD")? I sure have. In fact, I hate it so much that when we did OS/A+ for the Apple II, we allowed the user to supply a default device specifier (e.g., "D2:"), which is automatically prefixed to all file names which do not specify a device. (Consequence: you *must* use a colon when you really want a device; "P" is seen as "D2:P", though "P:" works fine.)

This concept is not new or unique; even in the micro world, such giants as CP/M use default drive assignments. Usually, the advantage of such defaults is that people with multiple disk systems need not always run a given program in a certain drive. Or the user might choose which drive will receive his data files via a simple set of keystrokes at system powerup. Suffice it to say that those who get used to default drives love them.

Unfortunately, as much as I would like to do the same thing for the Atari, I can't. The initial device name determination under Atari's OS is done in the OS ROMs, and Atari OS simply looks at the first letter of any file name and assumes that it is the device name.

However... (You knew there was a "however" lurking, didn't you?) At least we could modify the File Manager System (also known as FMS, DOS, or even OS/A+) to understand the concept of a default device NUMBER. In other words, we could have the FMS inspect the file name and assume a particular drive number if "D:..." were coded. Then we could have some means of telling

the FMS what the "current" drive was (and, in fact, such means already exist in OS/A+), and the system would automatically insert the correct drive number.

And yet, I am reluctant to adapt such an approach with Atari DOS. Too many programs have been written which assume that "D:..." is equivalent to "D1:...", and I am loath to introduce more confusion than is necessary. So, if you really would like to modify your copy (copies?) of FMS to allow "D:" to represent "Dn:", let me just point you in the right direction. For this purpose, I will presume that you have a copy of *Inside Atari DOS* (COMPUTE! Books, 1982).

There is a routine labeled FNDPCODE (File Name DeCODE) which begins on page 83 of the book and is the heart of the entire disk file name processing. Lines 4101 through 4106 start at the third character of the name and search from there backwards for the colon (':') which terminates the device specifier (and ignore the comments in the listing...they are flat out irrelevant). Obviously, it would be no big deal to check to see if the character before the colon is the 'D' and, if so, assign a default device number.

Changing FMS

Now, for the rest of you, I have an alternate proposal. How about changing FMS so that, if it sees a file name of "D0:..." it assigns the default device instead. I chose "D0:" because there should be no conflict with existing software. And, yet, it is a legal device specifier which is easily detectable and changeable.

Since the OS ROMs have already decoded the device number by the time FMS gets control, we don't need to look at the file name at all. Instead, we look at the field labeled ICDNO (or, in zero page, ICDNOZ), the device number as set up by the OS ROMs. And, conveniently, FMS is already manipulating this number in a single, well-defined place, the "SETUP" routine (as listed on page 92 of *Inside Atari DOS*). Currently, the code sequence is simply:

```
LDY ICDNOZ ; move device number...
STY DCBDRV ; ...to device control block
```

What we want instead is something like the following:


```

LDY    ICDNOZ    ;get device number...
BNE    OKDNO     ;...if wasn't "D0:", it is OK
LDY    DEFAULT   ;otherwise, change 0 to default
OKDNO
STY    DCBDRV    ;in either case, set up DCB

```

Now I can't think of a much simpler change than adding two instructions, but how do we make such a change? The solution is to use what is known as a "patch." Generally, there are two kinds of machine language patches: those that fit into the original code space and those that don't. The former kind are easy; simply overlay the old code with the new. The latter are not so easy. Naturally, this change falls into the latter category.

With a 6502, the usual method of installing out-of-line patches is to try to replace a three-byte instruction with a JMP or JSR to the patch (failing this, you must replace two or three instructions, which may involve putting a NOP before or after the JSR or JMP). Luckily, we do indeed have a three-byte instruction that we can replace (the STY DCBDRV uses three bytes, since DCBDRV is not in zero page).

So our patch will look like this:

```

DCBDRV = $301      ;object of the STY
*=      $1176      ;address of the STY instruction
JSR     PATCH
*=      PATCH
BNE     OKDNO      ;non-zero device number
LDY     DEFAULT    ;replace zero device number
OKDNO
STY     DCBDRV     ;the patched-over instruction
RTS

```

So far, so good. It makes sense, I hope. But there are two locations undefined in the above listing: we don't know where PATCH and DEFAULT are going to be located. Again, we will refer to the book for some clues as to where they should be.

As it turns out, there is no patch space at all within the main code space of FMS. However, if we look at the very end of the listing (page 98 in the book), we find that FMS (including its internal buffers, etc.) ends at \$1500. But remember that "DOS.SYS" consists of more than just FMS. In the case of OS/A+, DOS also includes "CP," the console processor, and actually ends at \$1D00. For Atari DOS, version 2.0S, DOS.SYS ends at \$1A7C (to accommodate "MINI-DUP," the routine which handles MEM.SAV and loads the main DUP.SYS).

But, fortuitously, whether by design or by chance, both MINI-DUP and CP begin at \$1540. Thus, we have locations \$1501 through \$153F for patch space. Not a huge patch space, but patch space nevertheless. So, I would suggest that you add the following two lines to the front of the listing given above:

```

DEFAULT = $1501
PATCH = $1502

```

This means, then, that you *must* put a valid disk drive number (1 through the number of drives you have) into location \$1501 *before* using a drive specifier of "D0:".

So, how do we make and save this patch? If you have an assembler capable of doing memory-to-memory assemblies (e.g., the cartridge, EASMD, MAC/65, etc.), I would suggest typing in the lines given and actually assembling the code directly in place. (Doing the memory-to-memory assembly avoids doing FMS accesses while patching FMS...safety first!) Then, with the patch in place, use the Write-DOS-Files option (of Atari DOS, or use INIT to rewrite DOS.SYS with OS/A+) to save your patched system.

Does it work? Sure does. I wrote all the above and then went over to the machine and typed it in. Worked first time! Is it handy? Only time will tell.

And one more point. If you do have OS/A+, you will note that the Command Processor (CP) already supports the concept of a default drive. Why not use that same default drive specifier for our "D0:" trick? The only difference is that CP stores that default specifier as an ASCII character ('1', '2', etc.), so we must look at only the low order bits of the default (and we must obtain it from its memory location according to OS/A+ rules). So here's another version of the same patch, specifically for OS/A+, version 2:

```

PATCH    = $1501
CPALOC    = $0A
DEFAULT   = 8
DCBDRV    = $301

*=        $1176
JSR       PATCH

*=        PATCH
BNE       OKDNO      ;drive # is non-zero
LDY       #DEFAULT   ;offset to default drive #
LDA       (CPALOC),Y ;gets default in ASCII
AND       #$0F       ;just the lower bits
TAY       ;where FMS expects drive #
OKDNO
STY       DCBDRV     ;the patched-over code
RTS       ;back to the original

```

And, as a postscript to all this, I would like to comment on the whole subject of adding things to DOS. So long as you can patch in place or use the limited patch space starting at \$1501, you should have no problems. If, however, you want to add significant code to DOS, it will not be easy if you are using Atari DOS.

If we look at pages 94 and 95 of *Inside Atari DOS*, we will see the routine which begins with the label "WD0". It is this routine which actually writes the file "DOS.SYS" to the disk. And, if you look at lines 5441 through 5449, you will see that what is written out is all of memory from \$7CB

through the contents of location "SASA" (which are usually \$1A7C or \$1D00, as noted above).

Sidelight: in a way, this is poor design, since SASA also specifies the beginning address of the disk buffers. If you move the disk buffers (e.g., to the top of memory) and then try to write the DOS file(s), you might be writing out much more than you bargained for. You might want to change those compares to

"CMP #..."

if you are doing hefty modifications.

Anyway, with Atari DOS, you can't really add on to the end of the DOS.SYS since DUP.SYS begins immediately after it in memory and would overwrite your additions. With OS/A+, though, you could add stuff at \$1D00 (or wherever SASA points to) and move SASA up (which not incidentally will thus move the buffers out of the way of your addition).

The Rites For Right Writes

I was reminded by all of the above of another "feature" of Atari DOS (and, yes, OS/A+) which is not well documented. In particular, would you like your program (including one written in BASIC) to be able to write (or rewrite) the "DOS.SYS" file? In the unlikely case that your answer is "yes," read on.

Strange but true: when you OPEN the file named "DOS.SYS" for output (i.e., mode 8 only), *right then and there* the FMS will automatically write the complete boot (sectors 1, 2, and 3) and the file "DOS.SYS" to the disk! You do *not* have to copy anything from memory to disk, from disk to disk, or what have you. FMS does it all! (And that explains why Atari DOS won't let you copy *to* a file called "DOS.SYS".)


Thus, from BASIC, you could initialize a disk AND write the DOS.SYS file via the following simple code:

```
10 XIO 254, #1, 0, 0, "Dn:"
20 OPEN #1, 8, 0, "Dn:DOS.SYS"
30 CLOSE #1
```

Of course, the "n" can be any valid disk number (including 0, if you applied the patches discussed in the first section of this column). Also, you can omit line 10 if you don't want to initialize the disk.

Unfortunately, this procedure will not place "DUP.SYS" on the disk if you are using Atari DOS, so you will still have to somehow copy it. (But you can use AUTORUN.SYS based systems without DUP.SYS, of course.) Again, though, if you are using OS/A+ you don't (and can't) use a DUP.SYS file, so the above little program will perform all you need to initialize a master, bootable disk.

Postscript: If you really *need* to copy a

"DOS.SYS" file from one disk to another (because, for example, you don't want to boot the version that you are copying), you can simply rename "DOS.SYS" to something else ("GORP.SYS", for example), perform the copy, and then rename both the old and new "GORP.SYS" back to "DOS.SYS". Thanks to the peculiarities of FMS, this method will even cause the three boot sectors to be updated to point to your new DOS file. 

**ATARI 400 48K
UPGRADE KIT**

- Uses your current memory board
- chips all guaranteed

99.95

- Prime quality 64k memory
- Add \$3.00 postage and handling

ATARI REFERENCE CARD


- Atari Basic commands
- All error codes
- All Peek/Poke locations

5.95

PRINTER REFERENCE CARD

- Printer control codes
- CITOH, EPSON, OKI DATA, NEC...

3.95


**CERMETEK 212A
300/1200 MODEM**


- Integral auto dialer
- Add \$5.00 postage and handling

560.00

MSX

Micro Systems Exchange
P.O. Box 4033
Concord, CA 94524
(415) 355-7130




**PAYROLL SOFTWARE
FOR
THE ATARI® 800™**

Miles Payroll System™ is an advanced and comprehensive payroll accounting system designed for businesses today. Cumulative totals are maintained for each employee, as well as complete reporting, check writing, and W-2 reporting. Some features include:

- Random access file organization for fast updating of individual records.
- Allows weekly, biweekly, semimonthly or monthly pay periods.
- Completely menu-driven and user-friendly.
- Regular, Overtime, Double time, Sick, Holiday, Vacation, Bonus and Commission earning categories.
- Payroll deductions include Federal W/H Tax, State W/H Tax, City W/H Tax, FICA, SDI, Group Insurance and 3 user-defined deductions.
- Tax sheltered annuity deduction capability for IRAs and other tax shelters.
- State and Federal Unemployment Insurance maintained.
- Complete file viewing and editing capability.
- Maintains up to 50 employees.
- Up to 10 user-defined Worker's Compensation classifications.
- Federal Tax tables may be changed in only 15 minutes each year by user when IRS changes tax.
- Table method used for State and City Tax, allowing compatibility with any state's or city's tax.
- Produces 15 different reports, including W-2 Forms Report.
- Checks calculated and printed automatically.
- PROGRAM ENABLING MODULE™ protects valuable payroll information from unauthorized users.
- 3 user-defined payroll deductions to accommodate customized needs such as savings, profit sharing, tax shelters, pensions, etc.
- Pay period, monthly, quarterly and yearly cumulative totals maintained for each employee.
- Automatic input error detection and recovery protects system from user-generated errors.
- Easy-to-follow, detailed, and comprehensive user's manual and tutorial leads the user step by step allowing anyone with little computer experience to easily operate the package.

Includes index.

- Color, sound, and graphics utilized for user ease.
- Maintains employee pay history.
- Allows for manual payroll check writing.
- Packaged in a handsome 3-ring deluxe pocketed binder with 3 diskettes and manual.
- Reasonable price.

See your local store, or contact Miles Computing.

Miles

MILES COMPUTING
7136 Haskell Ave. #204
Van Nuys, CA 91406
(213) 994-6279

Atari is a registered trademark of Atari, Inc.
Miles Computing, MILES PAYROLL SYSTEM, PROGRAM ENABLING MODULE are trademarks of Miles Computing, Van Nuys, California. Not affiliated with Atari, Inc.
\$179.95. Requires 32K and two Atari® 810™ disk drivers. Payment in U.S. funds required with order. California residents add 6.5% sales tax. C.O.D. or prepayment only. Dealer inquiries welcome.

Almost BASIC

This month we'll start a major project: a pseudo-BASIC interpreter written in Atari BASIC. Will this be a useful product? No. First, since it is written in and interpreted by Atari BASIC, it will of necessity be much slower than even Atari BASIC. Second, it will be an extremely limited language (as we'll shortly see) and, in fact, a nonstandard language.

But suppose we could overcome the first objection (speed) and ignore the second (so what if it is nonstandard, as long as it is ours). Would it be useful then? Sure. In fact, we could even speculate on rewriting the interpreter in C/65 or assembly language and ending up with an extremely fast, presumably integer-only interpreter. Still, the language is limited, and it would have to have some major extensions added before it would be really usable.

Enough speculation. Let's proceed to the language's definition.

1. The program editing scheme used will be essentially identical to that of Atari BASIC. Line numbers from 1 to some maximum will automatically be sorted and executed in order. Entering just a line number will erase any line with that number.
2. Single letter variables *only* will be allowed. This is a major point of departure from Atari BASIC, but it makes the interpreter significantly simpler. And no string variables.
3. Only the first letter of each statement name (command name) will be significant. Another big departure, and one which limits us to 26 different statements. Also note that this implies that if we use "Print," we can't use "Plot," "POKE," or "Position," etc. This also implies that you can keep programs small (and unreadable) by using single letter commands.
4. No functions. Sorry, but there will be no "RND(0)", no "SIN(30)", etc. This is necessary if we are to keep the expression analyzer down to manageable proportions when it is written in Atari BASIC.

5. No precedence of operators. Same excuse as number 4. This means that "3+4*5" will evaluate as "(3+4)*5" or 35. Most BASICs would see that as "3+(4*5)" or 60. Similarly, no parentheses will be allowed.

6. No provision for loading or saving programs. It would be easy to add this, and we might do so later. However, I see little point in doing so as long as the interpreter is running under Atari BASIC.

Whew! Feel restricted? Well, if you are adventuresome, you can try adding to and modifying the interpreter. It is a good exercise in logic, and you might even get good enough at it to give us a scare.

And one more thing before we get started with the heavy stuff. What do we call this thing? I haven't come up with anything better than BAIT, which is my acronym for BASIC (Almost) InTerpreter. (And which is also meant to imply that it is bait: I am fishing for innovation and interest from you, my gentle readers.)

BAIT Statements

Remember: only the first character of each statement/command name is significant, so what I am really presenting here is a list of which letters of the alphabet we are going to use. The table below lists the first letter, the mnemonic I am using, the syntax of the statement, and (in parentheses) the Atari BASIC equivalent, if indeed that BAIT statement is not the same.

A	Accept <variable>	(INPUT)
B	Begin	(RUN)
C	Call <line-number>	(GOSUB)
D	Display	(LIST)
E	End	
F	Fetch <address>, <variable>	(pseudo-PEEK)
G	Goto <line-number>	
I	If <expression>, <statement>	
L	Let <variable> = <expression>	
N	New	
P	Print <string-literal> Print <variable> Print	
R	Return	
S	Store <address>, <expression>	(POKE)

A few of the statements need explanation, which is given below. Also, note that line-numbers and addresses, as used in the above syntax, may

always be general expressions.

"Accept" allows only a single variable per use, unlike "INPUT" which allows several variables separated by commas.

"Fetch" and "Store" are complementary statements, both with the form of Atari BASIC's "POKE." The only difference is that "Fetch" obviously needs a variable (instead of an expression) to place the fetched (PEEKed) byte into.

"If" does *not* use a "THEN" keyword. Instead, any BAIT statement may follow the comma.

"Let" is a *required* keyword in BAIT. Actually, you may have already presumed this, since otherwise there is no way to distinguish a statement letter from a variable letter in such an assignment statement.

"Print" allows only one item to be printed per statement. Not shown in the above syntax, but allowed by BAIT, are the trailing semicolons or trailing commas, which have the same meaning as under Atari BASIC.

A discussion of what constitutes a valid expression, as well as several other more esoteric points, will have to wait for following month(s).

General Concepts

Since the code for BAIT will be presented in pieces over the course of several months, we must start with a coherent scheme. Also, since we will *not* reprint this month's code next month (for example), the listings must merge properly and neatly.

To this end, I have designated several line number ranges for specific purposes, as listed below.

1000-1999	Initialization of variables used as constants; dimension of strings and arrays; etc.
2000-2999	The "ready" prompt. Get a line of program/command. Parse line for line number.
3000-3999	Program editing. Delete and insert lines.
4000-4999	Control execution of running program. Execute next line, execute command line, etc.
5000-5999	Major subroutine which evaluates arbitrary arithmetic expressions by executing them.
8000-9999	Various miscellaneous subroutines, used by one or more statements.
10000 up	Execution of the actual statements and commands of BAIT. Line numbers of execution routine for each statement are defined in initialization segment, above.

Sidelight: What are the major differences between this scheme and that actually used by the authors of Atari BASIC? (1) There is no provision for generalized I/O routines. (2) Atari BASIC checks the syntax of each line as it is entered and tokenizes it into internal form right then and there. BAIT simply stores exactly what you type in. (3) BAIT is missing many, many of BASIC's capabilities, as noted above.

This Month's Listing

This program is my offering for this month. It consists primarily of the program editor, including the initialization need thereby.

One note about some temporary code: In the finished BAIT, lines 4000 through 4999 will control which statement/command will be executed next. In the case of a command (direct statement, in Atari parlance), these lines will pass control back to the ready prompt when the particular command executor returns. For program editing, we really only need one command, "Display" (LIST), so we have provided a very simple execute control which assumes that *all* direct statements are a request for "Display."

And now for some commentary on the code. Each section of comment is preceded by the line number (or range of numbers) that it refers to.

1010. I chose a practical number here. The larger MAXLINE is, the slower the line deletion process, and the larger the memory you will need. But feel free to change it.

1020. BUFFER\$ is used to hold the program you type in and can be almost any size, but be careful: I have not put any provisions in the current BAIT code for detecting when you run out of space.

1030. This is a departure from Atari BASIC (and an effective, though memory-consuming one). Rather than scanning through the program space (BUFFER\$) for a line, we "know" where it is via a table kept in LINES.

2360. Since I can't suppress the question mark which the INPUT on line 2300 produces, it is possible that using the Atari cursor keys will sometimes cause the "?" prompt to appear at the beginning of an input line. This gets rid of it by moving the right hand part of the string to the left. (It really works! Try it. And it's also used in line 2720.)

2520 and 2630. Remember, a completed FOR/NEXT loop exits with the loop variable already changed to the first failing value (thus LL + 1 in this example).

2710. If we don't do this, and if LP is greater than LL (i.e., if there is nothing following the line number), then the reference to LINES(LP) in line 2720 gives us a string length error.

3020. Necessary, if we stripped off the line number.

3040. Shame on you. You typed in a line number with a decimal point, trying to fool me. Gotcha.

3060. The only error message in this month's code.

3110. If the line doesn't yet exist, we can't delete it.

3120, 3130. The number stored in the "LINES"

table is the length of the line as stored in "BUFFERS" added to 1000 times its starting position in "BUFFER\$". We could have used two arrays (one for starting position and one for length) to make it neater, but it would have used a lot more memory.

3140. This line might not work, thanks to a bug in Atari BASIC. Perhaps next month we will have a fix to work around the bug. In the meantime, small programs in BAIT will always work. (Same as the my-system-went-away-when-I-deleted-a-line problem in Atari BASIC.)

3160-3180. This is tricky. After you remove a line via 3140, the starting position of all lines above it in the buffer must be adjusted downward by the size of the line deleted. Can you follow line 3170? Remember, "START" and "LENGTH" refer to the former start and length of the deleted line.

3210. In case we typed in just a line number.

3220-3240. Notice that each new line overlays the "*" which we tack onto the end of the buffer. We then have to put the "*" back on the end. This insures that line 3140 will always work properly, even when we delete the last line in the buffer.

3250. See the comments about lines 3120 and 3130.

3310. If it wasn't a direct line, assume it was added to the program and go after another line.

10100-10150. We check all possible line numbers to see if they need to be listed. Note the similarity between this code and the code needed to delete a line (lines 3110 through 3130): in both cases we need the starting position and length of the line.

10190. Note how each statement will simply RETURN to the execute control code.

Still with me? Go try it. Type it in *very* carefully, backing yourself up every 20 lines or so. If it doesn't work, go back and examine what you typed in, because I guarantee that it worked just seconds before I made this listing for **COMPUTE!**.

Next month, we will try our hand at adding Execute Expression (the most complicated part of what is left) and Print (so we can verify that expressions are executing).

```
1000 REM ..INITIALIZATION..
1001 REM .....
1010 MAXLINE=99
1020 DIM BUFFER$(5000),LINE$(128)
1030 DIM LINES(MAXLINE)
1040 FOR LP=0 TO MAXLINE:LINES(LP)=0:NEXT LP
1050 BUFFER$=""
1500 REM LINE NUMBERS OF EXECUTION ROUTINES
1510 PROMPT=2100:INNEXT=2300
1550 DODISPLAY=10100
2000 REM ..INTERACTION..
2001 REM .....
2100 PRINT "READY"
2300 INPUT LINE$
2350 IF LEN(LINE$)=0 THEN GOTO INNEXT
2360 IF LINE$(1,1)="?" THEN LINE$=LINE$(1):
```

```
GOTO 2350
2370 LL=LEN(LINE$)
2500 REM CHECK FOR LINE NUMBER
2510 FOR LP=1 TO LL
2520 IF LINE$(LP,LP)<="9" AND LINE$(LP,LP)>
  ="0" THEN NEXT LP
2550 REM LP HAS POSITION OF FIRST NON-NUMER
  IC CHARACTER
2560 CURLINE=0
2570 IF LP>1 THEN CURLINE=VAL(LINE$(1,LP-1))
2600 REM NOW SKIP LEADING SPACES, IF ANY
2610 IF LP>LL THEN 2700
2620 FOR LP=LP TO LL
2630 IF LINE$(LP,LP)=" " THEN NEXT LP
2699 REM
2700 REM REMOVE LINE NUMBER AND LEADING SPA
  CES
2710 IF LP>LL THEN LINE$="":GOTO 3000
2720 LINE$=LINE$(LP)
3000 REM ..EDITING..
3001 REM .....
3010 REM IF HERE, LINE NUMBER IS IN CURLINE
3020 LL=LEN(LINE$):REM AND LL IS LENGTH THE
  REOF
3030 IF CURLINE=0 AND LL=0 THEN GOTO PROMPT
3040 IF CURLINE<>INT(CURLINE) THEN 3060
3050 IF CURLINE<=MAXLINE THEN 3100
3060 PRINT "***BAD LINE NUMBER***"
3070 GOTO PROMPT
3100 REM FIRST, DELETE CURLINE IF IT ALREAD
  Y EXISTS
3110 LENGTH=LINES(CURLINE):IF LENGTH=0 THEN
  3200
3120 START=INT(LENGTH/1000)
3130 LENGTH=LENGTH-1000*START
3140 BUFFER$(START)=BUFFER$(START+LENGTH)
3150 LINES(CURLINE)=0
3160 FOR LP=1 TO MAXLINE:TEMP=LINES(LP)
3170 IF TEMP>=START*1000 THEN LINES(LP)=TEM
  P-LENGTH*1000
3180 NEXT LP
3200 REM NOW ADD LINE TO END OF BUFFER
3210 IF LL=0 THEN GOTO INNEXT
3220 START=LEN(BUFFER$)
3230 BUFFER$(START)=LINE$
3240 BUFFER$(LEN(BUFFER$)+1)="*"
3250 LINES(CURLINE)=START*1000+LL
3300 REM NOW LINE IS IN BUFFER...WHAT DO WE
  DO
3310 IF CURLINE THEN GOTO INNEXT
3320 REM *** TEMPORARY: JUST FALL THROUGH ~
  TO 4000 ****
4000 REM ..EXECUTE CONTROL..
4001 REM .....
4010 GOSUB DODISPLAY
4020 BUFFER$(INT(LINES(0)/1000))="*"
4030 LINES(0)=0
4040 GOTO PROMPT
4050 REM **** 4010 THRU 4050 ARE TEMPORARY ~
  ****
5000 REM ..EXECUTE EXPRESSION..
5001 REM .....
8000 REM ..MISCELLANEOUS SUBROUTINES..
8001 REM .....
10000 REM ..EXECUTE THE VARIOUS STATEMENTS..
10001 REM .....
10100 REM ==EXECUTE DISPLAY==
10110 FOR LP=1 TO MAXLINE
10120 LENGTH=LINES(LP):IF LENGTH=0 THEN 1015
  0
10130 START=INT(LENGTH/1000):LENGTH=LENGTH-1
  000*START
10140 PRINT LP;" ";BUFFER$(START,START+LENGT
  H-1)
10150 NEXT LP
10190 RETURN
```

OUTSIGHT: ATARI

April Meanderings

From Bill Wilkinson

There is so much to discuss this month, what with all the new announcements from Atari and others, that I won't waste your time with one of my cutesy little introductions, wherein I summarize — usually in one, long, run-on sentence (with many parenthesized asides) or one or two highly conjunctive paragraphs — all the things I might talk about this month, give away the punch line to various program listings, and apologize for mistakes made three or four months in the past, including mistakes that most readers never notice because (luckily) they just read the text and don't try to type in the program (or are clever enough to wait a month or two and see what mistakes I turn up in subsequent issues).

I will note that I will break from tradition a little this month and discuss some new software releases. I know that I have said before that I would not review software, but I feel that I must make an exception when it comes to new languages, especially those that come directly from Atari.

Atari COBOL

Unbelievable! After claiming for months (years?) that they were *not* after the business market, Atari did a complete about-face and produced an interpreter for that most popular business language: COBOL.

Although versions of COBOL have been available for a number of years in the CP/M market, as far as I know this marks the first attempt to implement it on a 6502. The implementation itself is revolutionary, also.

Most of my regular readers will no doubt recall that I have repeatedly stated that the 6502 is a lousy machine to write a compiler for. Several groups have attempted to solve this problem by producing code for an arbitrary "p-machine" and

then writing a p-code interpreter which emulates the (possibly imaginary) p-machine. Examples of this type of compiler include Atari Pascal, UCSD Pascal for the Apple, and (in a similar but not identical vein) Forth.

Now, p-code is small, and the p-code interpreter can be fairly compact and efficient. But a COBOL p-code interpreter (should we call it C-code?) would be fairly large, because of the great variety of data types, etc., that COBOL supports. So why bother with the compiler stage? Why not tokenize the user's input, à la Atari BASIC, and directly interpret the tokens? You save a lot of space and sacrifice only a little bit of speed. Voilà.

Anyway, I recognize that not too many **COMPUTE!** readers are COBOL aficionados, so let's do a very short exploration of COBOL in general and Atari COBOL in particular. Insofar as possible, I will try to relate COBOL features to BASIC features.

COBOL programs are always divided into four major *divisions*: the identification division, the environment division, the data division, and the procedure division.

There really is nothing in BASIC to correspond to either the identification or environment divisions. The identification division is a kind of forced REMark section; its contents are usually installation and/or compiler specific. Under Atari COBOL, this division is used to specify programmer name, date of compile, and auto-boot procedures (if any). The environment division is used by the COBOL programmer to tell the compiler about the hardware configuration that the compiled code is *destined* for. Atari COBOL allows the user to specify whether he or she is running on a 400, 800, or 1200 computer and describe the memory configuration. One can also specify whether a

color TV or a black and white monitor is in use, using either the American NTSC system or the European PAL scheme. Naturally, one can describe the kind of printer to be used (daisy wheel, dot matrix, etc.).

COBOL's data division is at best poorly translatable to BASIC. With COBOL, one must

Wisely, Atari chose to disallow all use of the procedure division, ...

declare all variables, while BASIC users declare only strings and arrays. Also, with COBOL, each variable has a "PICTURE" associated with it which specifies what the data the variable represents will look like when it is displayed or printed (kind of like having a built-in "PRINT USING" format for each variable). Of course, some variables are never printed or displayed, so they can be declared "COMPUTATIONAL," without a picture associated.

Atari COBOL expands on the "PICTURE" concept by allowing the user to specify graphic modes and pictures, including the capability of declaring a variable to be a "player" or "missile" which will be automatically animated (moved both horizontally and vertically) during the VBLANK interrupt period. Of course, there is a price to pay for this flexibility: just setting up a player can require as many as 48 lines of code!

The final division (and, yes, the divisions must be kept in proper order in a COBOL program) is the procedure division. It is here that COBOL looks the most like other languages, including BASIC. There are COBOL equivalents to many BASIC statements, including GOSUB ("PERFORM"), LET ("MOVE"), IF...THEN ("IF"), and several more. Obviously, all the useful "work" of COBOL is done in the procedure division.

While COBOL does not use line numbers, it suffers some of BASIC's problems; and the user must work to write properly structured COBOL programs. Perhaps the real beauty of COBOL is its ability to be extremely self-documenting. How much more readable it is to say "IF SALES GREATER THAN QUOTA MOVE BONUS-AMOUNT TO BONUS IN SALARYRECORD". And that really is legitimate COBOL code!

The most fantastic aspect to Atari COBOL is that somehow Atari managed to fit the whole thing into an 8K byte cartridge. Rumor has it that they have developed a 16-bit virtual machine that does the brunt of the work. (I don't believe the rumor that Atari wrote COBOL in BASIC and is going to call the manual "An Introduction to BAS-BOL." On the other hand, who can say?)

The one unfortunate aspect to Atari COBOL is that, in order to cram it all into the cartridge, they had to omit one of the four major divisions. Wisely, Atari chose to disallow all use of the procedure division, since they felt that all but the most experienced COBOL programmers would not miss it.

Is Something Unclear?

I received a letter from Y. D. Obon, of Erehwon, Nebraska, regarding a comment I had made many months ago about suppressing the screen clear when changing graphics modes. I had said at the time that there seemed little use in such a capability. Well, once again, I have been proved wrong.

Since my comment appeared a long time ago, and since it was written in connection with my assembly language graphics library, I will restate it in terms of Atari BASIC. If the GRAPHICS statement had been omitted from Atari BASIC, the user would have been able to perform the equivalent function by typing in the following equivalent statements:

```
CLOSE #6  
OPEN #6, 12 + n, m, "S:"
```

In that second line, "m" is the graphics mode (e.g., m = 7 is equivalent to GRAPHICS 7). Also, "n" is 0 if full screen graphics are desired, and 16 if you want four text lines at the bottom of the screen. BASIC generates "n" for you based on the GRAPHICS mode you select; note that BASIC inverts the sense of the "+ 16" before performing the OPEN. (Note that the "12" is simply to tell CIO that we can do both input and output on this channel. It is not used by "S:".)

However, "n" as shown above can take on at least one other meaning besides selecting full screen or text mode graphics. If "n" equals 32 (or 48), the screen clear which usually takes place upon changing of graphics modes is suppressed. Now I hadn't thought this feature of much use. After all, if I had a mode 5 graphics display and attempted to change to mode 6 without clearing the memory, I would get some sort of meaningless jumble on the screen.

The program demonstrates several points, including that made in the previous paragraph. I was sorely tempted to simply dump this listing on you, without explanation, and let you try it out. But I will take pity. At the very end of this

month's column, there is an explanation of the various points made by this program. *Please, please don't read it yet!* Please type the program in and run it first. One clue: not only is the program a lesson in and of itself, but it also reveals the main point of this month's column up to this point.

Caution: Double and triple check the program before you RUN it. The effect could be disastrously scrambled if you make a mistake. On the other hand, the listing is short enough that you should be able to type it in error-free.

Confusion Is Good For A Sole

Actually, maybe I'm a heel instead of a sole. But I had to have a *little* fun with this column, and April is obviously the month for it.

I have had a few people pre-read this month's column, and the consensus is that I should explain some of my jokes. Now, I have always felt that a joke that has to be explained isn't funny, but maybe practical jokes are exceptions.

I would first like to point out that even while I was pulling your assorted limbs, I was trying to give you good and valid information. In the discussions of COBOL, everything I told you was accurate and truthful, *except*, of course, anything that referred to Atari. For instance, COBOL really does have four major divisions, and it really does support a PICTURE capability in the data division. But I think you will find Atari buying IBM (the whole company, not just the computers) before you will find them producing a COBOL for any of their current crop of machines.

Finally, the "kicker" in the description of Atari COBOL (the giveaway that it was all a joke) was the statement that the procedure division was not supported. That is roughly equivalent to leaving *all* statements out of Atari BASIC other than REM, DATA, and DIM!

A Slight Hand

If you have not yet typed in and RUN the BASIC program at the end of this article, you are cheating if you start reading this part. Shame, shame, on you. Actually, the program is a kind of puzzle. How and why does it act as it does? Well, the easiest way to explain is to discuss it, line by line.

Line 10: There's nothing special about the name of the string, DIM\$. I chose that name just to show that keywords are *not* generally reserved names in Atari BASIC. GRAPHICS 23 is actually GRAPHICS 7 + 16, full screen mode 7 graphics. Note that this statement will clear all screen memory used in mode 7.

Line 20: We're going to do a loop 40 times. We READ a character from the DATA statements and use its ATASCII value as a COLOR. Trick: only the two least significant bits of a color number are used in mode 7. Thus COLOR 3 is the same as

COLOR 7 is the same as COLOR ASC("G"), because ASC("G") is 71.

Line 30: We draw some nice vertical lines, each in a color determined via the READ in line 20. Remember, plotting points on an Atari really means we are turning on or off certain bits in the computer's screen memory. Isn't this a peculiar set of bit patterns?

Line 40: Naughty, naughty, Bill. You used an XIO 12 instead of a CLOSE! Yes, but the point is that doing a CLOSE from BASIC really is the same as an XIO 12. So we closed IOCB #6, the screen device. And what about the rest of the stuff, the "237,91,"* = LABEL"? Junk. Pure junk. It is totally ignored by CLOSE, and is meant only to mislead you.

Line 50: More of the same foolishness. XIO 3 is exactly the same as an OPEN command. We are opening the screen ("S..." is the screen, the "... " are ignored, natch) on IOCB #6 (which is where GRAPHICS would open it). We choose graphics mode 1 (the second parameter), and the 44 may be thought of as 32 + 8 + 4. The 32 says *don't* clear the screen memory, 8 says we can write to the device (the screen), and 4 says we can read from it (though we don't in this program).

Line 60: So that we can leave the full screen graphics active.

Lines 70, 80, and 90: As explained above, only the two lower bits of each of these characters are used. We could have used 2,2,4,5, etc., instead, but I worked to get these in alphabetical order, to confuse you further!

And why does it work like it does? Because we are actually seeing the stuff we plotted in mode 7 in a different way. Those same bits which were used in pairs as colors are now interpreted as bytes of eight bits each which are seen as characters.

So now you know the secrets. But you still can't see the surprising result unless you take the time to type in and run the program. Which you already did. Unless you cheated.

Well, Henny Youngman I'm not, but I hope you enjoyed this month's foolishness. Next month, on to more serious things. Finally, we will start showing how to write self-relocatable assembly language. Until then, best wishes from the Lo Of Lirpa.

```
10 DIM DIM$(1):GRAPHICS 23
20 FOR X=20 TO 59:READ DIM$:COLOR ASC(DIM$)
30 PLOT X,0:DRAWTO X,91:NEXT X
40 XIO 12,#6,237,91,"* = LABEL"
50 XIO 3,#6,44,1,"SAVE D:TEST"
60 GOTO 60
70 DATA B,B,D,E,H,K,L,L,N,O,P,R,T
80 DATA B,B,E,J,J,K,L,L,L,L,N,N
90 DATA A,B,D,F,G,G,J,J,K,K,L,N,O,P
```

INSIGHT: Atari

Bill Wilkinson

The series on writing your own interpreter continues. In part 2, the expression evaluator and the "PRINT" statement are added to BAIT. There's also a look at Atari's new 200XL computer.

We hope to introduce several new products at the West Coast Computer Faire this year, including some designed specifically for the new model 1200 Atari (of which machine I will speak more below). I can't tell you exactly what the new products will be, but I can say that I think that those who have written software which follows the "rules" will benefit.

Which "rules"? Oh, nothing much. Just those regarding LOMEM, HIMEM, device drivers, reset vectors, break vectors, etc. If you are an author (or company) who is developing or has developed software for the Atari computers, you might want to ask Atari for a copy of the note from Howard Chan, Manager of Software Acquisition, which details what Atari considers the "untouchable" locations as well as what "vectors" are immutable. We hope to be able to reproduce that note in this column next month.

Anyway, what are we looking into in this month's column? Obviously, we will have part two of the series on writing your own interpreter. (And if you missed part one, you must go out right now and buy the March issue! We cannot and will not recap the materials previously covered.) Also, as mentioned, I would like to briefly discuss the new Atari 1200XL machine. But first I am going to hang my head a little.

Pardon Me, My Prafall Is Showing

After giving everyone else (particularly Atari) a hard time about not doing things "right," I am embarrassed to admit that I, too, did a thing definitely "un-right."

I must start by giving credit to F. T. Meiere, President of the Indy Atari Club from Indianapolis, for not only finding my goof, but also giving me what seems to be a workable and proper fix.

The mistake occurred, not surprisingly, in my fix to the Atari RS-232 drivers, as published in this column in the December 1982 issue of **COMPUTE!**. It came about because of the variety

of configurations that I work in. The possible combinations I use can be shown as a small array:

	Atari DOS 2.0s	OS/A + version 2	OS/A + version 4
Cartridge Software			
RAM-based Software			

Now, obviously, the vast majority of the Atari user population finds itself in the upper left box (Atari BASIC with Atari DOS). And, yet, because I really don't like working with "MEM.SAV" and "DUP.SYS" (and the consequential swapping in and out and sometimes losing my memory and ...), I generally leave that left-hand column for last. And, unfortunately, in this case I apparently didn't even get to it. For shame.

Anyway, taking F.T. Meiere's advice to heart, I have indeed tested the change he has proposed in several of the possible configurations. Additionally, I have looked at my original code and found out why it failed (and why this new code works). So here, without further ado, is the fix to my RS-232 fix in the form of a change to line 1990 of the assembly language code:

was: 1990 JMP (DOSINI) WRONG!
now: 1990 JMP PATCH3 RIGHT!

To Excel Or Not To Excel

The new Atari machine is named the "1200XL." I suppose the "XL" is supposed to designate speed and sexiness, à la sports cars. And certainly the machine *looks* sleek and sexy enough; it is by far the best looking of the current crop of home computers. Were it not for the serial I/O cable, you could easily envision holding the machine in your lap while leaning back in your easy chair, admiring and caressing it as you would a glass of good wine.

Let's look at the obvious features:

- *Pluses:* 62K of RAM, two character sets, a self-test

capability, nearly complete compatibility with the 400/800 systems, four function keys and a "help" key, two status LEDs.

- *Minuses:* One cartridge slot (on the side, and you *can* remove the cartridge with power on even though you shouldn't), two (not four) joystick ports (both on the same side of the case; consider getting a joystick cord extender for two-person games), no memory board slots, no external expansion capabilities.

- *Implications:* Goodbye, 80-column boards. Goodbye, RAMDISKS and the like. Goodbye, CORVUS hard disk drive (which, I believe, interfaces via joysticks three and four).

- *Unfounded rumors:* There is *not* an RS-232 interface built in. There is certainly *no* parallel printer port. In fact, there is no hardware other than what I have described.

Some "features" of the machine are less obvious: none of the current Atari software will take advantage of the expanded RAM. When you bank select the RAM, all of the OS software, including the interrupt handlers, goes away, so you must provide at least a minimal OS substitute. Because the I/O space is from \$D000 to \$D800 (as on the 400/800), there is no way around having a "hole" in your otherwise contiguous RAM. There is no way to get at the RAM which is "under" the cartridge (this flaw is left over from the 400/800; it is a real deficiency). It uses the same old slow floating point routines.

So how do I rate the 1200XL in overall features and performance? Quite honestly, it depends entirely on what the price of the machine is. At anything under \$450, it's a terrific bargain. I feel that, given the obvious cost-cutting Atari was able to achieve, it should be able to sell for half the cost of the 800. However, the indications are that the price of the 800 will be dropped and that the 1200 will cost more than the 800. If so, buy an 800 quick!

The exception to this suggestion is if you will write in machine language or be using non-Atari languages that can take advantage of the extra 14K of RAM (now *where* would you get a language like that?). If you *need* the extra RAM, then you may have to seriously consider the 1200. Of course, by the time you read this, the price of the 1200 and the new price of the 800 should be public knowledge, so you will be able to see how accurate my forecasting is.

BAIT, Part 2

In March, we started the process of writing a pseudo-BASIC interpreter, which I called "BAIT." If you don't have that article, this month's work will make virtually zero sense, so don't even attempt to follow the rest of this column.

This month, as promised, we add the expression evaluator and the "PRINT" statement to BAIT. Note that the listing published here is *not* complete. It is meant to be added to the March listing. In a few cases, this month's lines will overwrite (be the same number as) those from March. For example, we have replaced lines 4010 through 4040 and deleted line 4050.

Before we get into the explanation of the actual listing, we need to extend our discussion of just how an interpreter – and, in particular, BAIT – works.

There are two major parts to most language interpreters: the program editor and the program executor. The March column presented BAIT's editor. It is not fundamentally different from most BASIC editors. True, only a few BASICs that I know of use a line number table, as we did for BAIT (some that do include Cromemco 32K Structured BASIC, which we wrote, and Data General's Business BASIC, both designed for relatively large machines). But, to be fair, BAIT cheats by using a very small fixed number of possible line numbers.

The editor used by Atari BASIC and BASIC A+ (and Cromemco and DG BASICs) does, however, differ markedly from BAIT's editor in one important aspect. In these more sophisticated BASICs, the user's program line is scanned for correct syntax as it is entered and automatically converted to more usable internal "tokens." Of course, BAIT should not be chided for any deficiency here: most microcomputer BASICs (including, for example, Microsoft BASICs) do *not* do any syntax checking at entry (nor do they tokenize anything except, perhaps, recognized keywords). In any case, BAIT's editor seems quite adequate to me.

This month, we begin the second major part of an interpreter: the program executor. Not surprisingly, the program executor is much larger and more complex than the editor. In fact, we need to break the executor down into manageable hunks. I think an outline would be useful here.

I. Program Editor

II. Program Executor

A. Initialization

B. Execution by Line

1. Execution by Statement

2. Execution of Statements

a. Display statement

b. Print statement

... (various statements)

C. Execution of a direct statement or line

D. Error handler

This month, we will add parts C, D, and B to BAIT. (Note that we did part A in March and faked C.) Actually, part C and part B are so inti-

mately entwined in BAIT that it is hard to see where one begins and the other leaves off, but that doesn't make our outline any less valid.

Executing Expressions In BAIT

Not shown in the above outline are the major routines which are common to the execution of most statements. To illustrate, first consider these two BAIT statements:

```
L A = 7*13      (Let A = 7*13)
P A + 5         (Print A + 5)
```

What do these two statements have in common? An expression. From BAIT's viewpoint, the two expressions here are "7*13" and "A + 5". A major portion of BAIT (and, indeed, a major portion of *any* language) is the subroutine known as "EXecute EXpression," which resides in lines 5000 through 5999 in the accompanying listing. Actually, EXEXP in BAIT is fairly simple when compared to that of Atari BASIC. Remember the rules from last month? No functions, no precedence of operators, no arrays, no strings.

Not surprisingly, almost all BAIT statements call the EXEXP subroutine. In turn, EXEXP calls a couple of routines, including GETNC (GET Next Character - lines 8100 to 8160). GETNC is perhaps the lowest level routine of the program execution phase of BAIT. It simply scans the program memory for the next non-space character, tests to see if it is an alphabetic character, and protests when the line runs out of characters.

EXEXP uses GETNC (line 5100) to find any ALPHAbetic characters in an expression; such characters are assumed to be variables (lines 5300, 5310). If instead, GETNC found a numeric character (line 5110), EXEXP backs up and scans for the entire number (lines 5400 to 5450). Only digits and a decimal point are allowed (line 5430); but there is a flaw (read that as *bug*) here that allows, but ignores, more than one decimal point and the digits which might follow. Finally, if the character is neither alphabetic nor numeric, BAIT assumes that it is an operator and figures out which one (lines 5120 to 5230). If it is not an operator, and if the expression was valid, EXEXP returns to its caller (line 5160).

Note that in the case of either a variable or a numeric literal, EXEXP assumes that it has received the second argument of an expression of the form "arg1 op arg2" (lines 5500 through 5530). Of course, in the case of the very first argument in any expression, there has been no preceding argument. But EXEXP takes care of that by providing a dummy argument ("0") and a dummy operator (" + ") in its initialization code (line 5010). Incidentally, if EXEXP detects two operators or two arguments in a row, it rules the expression invalid (lines 5210, 5220, and 5510). Similarly, null

expressions and expressions ending in an operator are illegal (lines 5230, 5530, and 5160).

Finally, the actual operators of BAIT are "simulated" via Atari BASIC in lines 5610 through 5680. Note that BAIT allows BASIC's operators "+", "-", "*", "/", ">", "<", and "=" . BAIT simplifies the inequality sign to "#", instead of BASIC's "<>". (But did you know that many, many of the early BASICs used or allowed "#" as an alternative to "<>")

Normally, I wouldn't be so bold as to suggest changing an entire section of code, but I think the clumsiness of EXEXP deserves at least one alternative idea. If you are using BASIC A+ (or any BASIC with a "FIND" or "SUBSTRing" function), you could replace lines 5120 to 5128 with a single line of code:

```
5120 OP = FIND( " + - * / > < = #", CS, 0 ) : IF OP
      THEN 5200
```

Of course, one could have achieved similar results with a string and a FOR/NEXT loop under Atari BASIC, but that would have slowed down EXEXP even more than it already is.

BAIT's Print Statement

Lines 10200 through 10330 comprise the execution of "Print" under BAIT. Notice that DOPRINT also uses GETNC (line 10210). Here, we are looking to see whether a quoted string (line 10220), an expression (line 10240), or nothing at all (line 10210) follows the "P" keyword. (Or should we call it a key-letter?)

Literal strings are fairly simple to handle. Starting at the character after the quote mark, we simply loop through the buffered line printing characters as we go and looking for an ending quote (lines 10300 and 10310). If no matching quote is found, it is *not* an error, just as with Atari BASIC (end of line 10310). If the quote is found, we adjust the character pointer and look for a trailing semicolon or comma (lines 10320, 10330, then 10250 to 10280).

And, strangely enough, arithmetic expressions are the easiest of all things to print. We simply call EXEXP and display the calculated result (line 10240), falling through to the trailing semicolon and comma check. (Of course, if we were writing in assembly language, we would have to write the "display a numeric result in ASCII" routine, but even here the Atari OS ROMs would help us.)

What Else Was Added

Finally, we must comment on the other code that was added this month. Most of it, of course, was needed to support the EXEXP and DOPRINT routines. However, some of it certainly is obscure enough to bear explanation. As we did in March, we will comment on the code by line number(s).

1100. C\$ is used to capture the next character by GETNC. The array VARIABLES is designed to hold 26 variables (A-Z). One could easily amend this to any multiple of 26 and allow variable names of the form A1, A2, etc.

1110. This is kind of silly. In the final code, all variables will be initialized to zero. However, since we do not yet have a "Let" statement, I wanted to give each variable a unique value so we could use it in "Print". Hence, A=1, B=2, C=3, etc.

1120. Simply a place to stuff an error message.

1520 to 1550. The line numbers of some of our more important routines.

1710. I hate using "TRAP 40000". I like "TRAP UNTRAP" much better.

2360. The only line I actually corrected from the March listing. Do you see what the bug was?

3320. Just changed the comment to make more sense.

4010 to 4040. The beginnings of our "Line execution" control routine. We get the starting and ending positions of the current line. If the line doesn't exist, we try for the next line. If this is a direct line, we flag it for later detection (line 4040).

4210. As things sit now, if we get here we are ready to execute the direct statement. It had better be the "P" (Print) key-letter.

4220. Why call line 4900? Why not do it in-line right here? Wait until next month.

4610. If we didn't just execute a direct line, we go do another line. (Won't happen this month.)

4620 to 4640. This code was at lines 4010 to 4040 last month. It just cleans up the program buffer for use by the editor.

4910. Read line 4920.

5010 to 8160. Described in the text above.

8200 to 8290. Why do this several places when a single routine will do? Note line 8240: Atari BASIC does a similar thing with the 6502's CPU stack when it encounters an error. Why try to recover through who knows how many sub-routine calls when one can simply reset the stack to the top and ignore them?

10200 to 10330. Described in the text above.

Using What We Have

Again, BAIT seems to work as designed up to this point. You can type in program lines (with preceding line numbers) or you can type in a direct statement. Unfortunately, all direct statements are assumed to be "Print," but just wait until next month.

And just what can you "Print"? Virtually any numeric expression that uses the BAIT operators and literal numbers. Of course, you can also use

the variable letters "A" through "Z," but this month you will get the artificial values they contain. To get you started, here are some statements to try when you get BAIT's "ready" prompt:

```
P "HI THERE"
P "HI THERE",
P "HI THERE";
P 1+2+3+4
P 1 + 2 + 3 + 4
P A+B+C+D
P 4*5
P 4<5
P 1/3
P 1/2=0.5
P 1/2 # 0.5
P 1/3;
```

And one last P.S., a kind of taste of what's to come. Once you have the listing working and saved, try adding one line:

```
4905 IF C$="D" THEN GOTO DODISPLAY
```

If you don't see what it allows, then wait for next month.

Next Month

Naturally, we will have Part 3 of BAIT. We will actually begin running BAIT programs, and we will add about half of the remaining BAIT statements to our vocabulary.

Unless something else hits me in the next week or two, I think I will respond to my own challenge and begin talking about how to write self-relocatable assembly language.

```
1100 DIM C$(1),VARIABLES(26)
1110 FOR ALPHA=0 TO 26:VARIABLES(ALPHA)=ALPHA:NEXT ALPHA
1120 DIM ERR$(40)
1520 LET GETNC=8100
1530 SYNTAX=8300:ERROR=8200:EXEXP=5000
1550 DODISPLAY=10100:DOPRINT=10200
1700 REM MISCELLANY
1710 UNTRAP=40000
2360 IF LINE$(1,1)="?" THEN LINE$=LINE$(2):GOTO 2350
3320 REM NOTE THAT CURLINE=0 AS WE FALL TO LINE 4000
4010 LENGTH=LINES(CURLINE):IF LENGTH=0 THEN 4600
4020 CURLOC=INT(LENGTH/1000):LENGTH=LENGTH-1000*CURLOC
4030 CUREND=CURLOC+LENGTH-1
4040 IF CURLINE=0 THEN CURLINE=-1
<<< DELETE LINE 4050>>>
4100 REM READY TO EXECUTE A LINE
4200 REM EXECUTE THE STATEMENT
4210 GOSUB GETNC:IF NOT ALPHA THEN GOTO SYNTAX
4220 GOSUB 4900
4600 REM COME HERE FOR NEXT LINE
4610 CURLINE=CURLINE+1:IF CURLINE>0 THEN 4000
4620 BUFFER$(INT(LINES(0)/1000))="*"
4630 LINES(0)=0
4640 GOTO PROMPT
4900 REM THE STATEMENT CALLER
```



```

4910 GOTO DOPRINT
4920 REM LINE 4910 IS TEMPORARY !!!!
5010 EVAL=0:LASTOP=-1
5020 VALID=0
5100 GOSUB GETNC:IF ALPHA THEN 5300
5110 IF C$>="0" AND C$<="9" THEN 5400
5120 REM WHICH OPERATOR?
5121 IF C$="+" THEN OP=1:GOTO 5200
5122 IF C$="-" THEN OP=2:GOTO 5200
5123 IF C$="*" THEN OP=3:GOTO 5200
5124 IF C$="/" THEN OP=4:GOTO 5200
5125 IF C$=">" THEN OP=5:GOTO 5200
5126 IF C$="<" THEN OP=6:GOTO 5200
5127 IF C$="=" THEN OP=7:GOTO 5200
5128 IF C$="#" THEN OP=8:GOTO 5200
5160 IF VALID THEN RETURN
5170 GOTO 5900
5200 REM GOT AN OPERATOR
5210 IF LASTOP>0 THEN 5170
5220 IF LASTOP<0 AND OP>2 THEN 5170
5230 LASTOP=OP:VALID=0:GOTO 5100
5300 REM GOT A VARIABLE
5310 VAL2=VARIABLES(ALPHA):GOTO 5500
5400 REM GOT A NUMERIC
5410 CURLOC=CURLOC-1:REM BACKUP TO FIRST N
    UERIC
5420 FOR LL=CURLOC TO CUREND:C$=BUFFER$(LL
    )
5430 IF (C$>="0" AND C$<="9") OR C$="." TH
    EN NEXT LL
5440 VAL2=VAL(BUFFER$(CURLOC,LL-1))
5450 CURLOC=LL
5500 REM VAR OR NUMERIC
5510 IF LASTOP=0 OR ABS(LASTOP)>8 THEN 590
    0
5520 GOSUB 5600+10*ABS(LASTOP)
5530 LASTOP=0:VALID=1:GOTO 5100
5600 REM EXECUTE OPERATORS
5610 EVAL=EVAL+VAL2:RETURN
5620 EVAL=EVAL-VAL2:RETURN
5630 EVAL=EVAL*VAL2:RETURN
5640 EVAL=EVAL/VAL2:RETURN
5650 EVAL=(EVAL>VAL2):RETURN
5660 EVAL=(EVAL<VAL2):RETURN
5670 EVAL=(EVAL=VAL2):RETURN
5680 EVAL=(EVAL<>VAL2):RETURN
5900 ERR$="INVALID EXPRESSION":GOTO ERROR
8100 REM GETNC
8110 IF CURLOC>CUREND THEN C=-1:C$=CHR$(15
    5):GOTO 8140
8120 C=ASC(BUFFER$(CURLOC)):C$=CHR$(C)
8130 CURLOC=CURLOC+1
8140 IF C=32 THEN GOTO GETNC
8150 ALPHA=(C$>="A" AND C$<="Z")*(C-64)
8160 RETURN
8200 REM ERROR ROUTINE
8210 PRINT:PRINT "****";ERR$;"****";
8220 IF CURLINE>0 THEN PRINT " AT LINE ";C
    URLINE
8230 PRINT:TRAP 8250
8240 POP:POP:POP:POP:POP:POP:POP:PO
    P
8250 TRAP UNTRAP
8290 GOTO PROMPT
8300 REM SYNTAX ERROR
8310 ERR$="SYNTAX ERROR":GOTO 8200
8320 REM ==EXECUTE PRINT==
10210 GOSUB GETNC:IF C<0 THEN PRINT:RETURN
10220 IF C=34 THEN 10300
10230 CURLOC=CURLOC-1
10240 GOSUB EXEXP:PRINT EVAL;

```

```

10250 IF C$=";" THEN RETURN
10260 IF C$="," THEN PRINT:RETURN
10270 IF C<0 THEN PRINT:RETURN
10280 GOTO SYNTAX
10300 FOR LL=CURLOC TO CUREND:C$=BUFFER$(LL
    )
10310 IF ASC(C$)<>34 THEN PRINT C$;:NEXT LL
    :PRINT:RETURN
10320 CURLOC=LL+1:GOSUB GETNC
10330 GOTO 10250

```

Use the handy
reader service cards
in the back of the
magazine for
information on
products advertised in
COMPUTE!

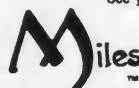


PAYROLL SOFTWARE FOR THE ATARI® 800™

Miles Payroll System™ is an advanced and comprehensive payroll accounting system designed for businesses today. Cumulative totals are maintained for each employee, as well as complete reporting, check writing, and W-2 reporting. Some features include:

- Random access file organization for fast updating of individual records.
- Allows weekly, biweekly, semimonthly or monthly pay periods.
- Completely menu-driven and user-friendly.
- Regular, Overtime, Double time, Sick, Holiday, Vacation, Bonus and Commission earning categories.
- Payroll deductions include Federal W/H Tax, State W/H Tax, City W/H Tax, FICA, SDI, Group Insurance and 3 user-defined deductions.
- Tax sheltered annuity deduction capability for IRAs and other tax shelters.
- State and Federal Unemployment Insurance maintained.
- Complete file viewing and editing capability.
- Maintains up to 50 employees.
- Up to 10 user-defined Worker's Compensation classifications.
- Federal Tax tables may be changed in only 15 minutes each year by user when IRS changes tax.
- Table method used for State and City Tax, allowing compatibility with any state's or city's tax.
- Produces 15 different reports, including W-2 Forms Report.
- Checks calculated and printed automatically.
- PROGRAM ENABLING MODULE™ protects valuable payroll information from unauthorized users.
- 3 user-defined payroll deductions to accommodate customized needs such as savings, profit sharing, tax shelters, pensions, etc.
- Pay period, monthly, quarterly and yearly cumulative totals maintained for each employee.
- Automatic input error detection and recovery protects system from user-generated errors.
- Easy-to-follow, detailed, and comprehensive user's manual and tutorial leads the user step by step allowing anyone with little computer experience to easily operate the package. Includes index.
- Color, sound, and graphics utilized for user ease.
- Maintains employee pay history.
- Allows for manual payroll check writing.
- Packaged in a handsome 3-ring deluxe pocketed binder with 3 diskettes and manual.
- Reasonable price.

See your local store, or contact Miles Computing.



MILES COMPUTING
7136 Haskell Ave. #204
Van Nuys, CA 91406
(213) 994-6279

Atari is a registered trademark of Atari, Inc.
Miles Computing, MILES PAYROLL SYSTEM, PROGRAM ENABLING MODULE are trademarks of Miles Computing, Van Nuys, California. Not affiliated with Atari, Inc.
\$179.95. Requires 32K and two Atari® 810™ disk drivers. Payment in U.S. funds required with order. California residents add 6.5% sales tax. C.O.D. or prepayment only. Dealer inquiries welcome.

INSIGHT: Atari

Bill Wilkinson

This month Bill continues with the creation of the BAIT interpreter (Basic Almost InTerpreter). And he includes some comments from readers.

BAIT: Part 3

For those of you who may have missed Parts 1 and 2, let me give a brief description of this project. BAIT is an acronym for Basic Almost InTerpreter. It is a pseudo-BASIC actually written in Atari BASIC. It is slow. It uses one letter commands (for example, "P" for PRINT). It is simple. And its purpose is simply to give you an inkling of how a BASIC interpreter works. It is *not* a finished, usable language.

This month we will study Part 3 of this listing. We will publish only those lines which have changed from Parts 1 and 2. However, next month we will present Part 4, the last part, and we'll publish the entire listing.

Before starting on my own comments about and additions to BAIT this month, though, I would like to share some reader comments on Part 1.

First, Howard Fishman of Brooklyn, New York, pointed out that I could eliminate the question mark prompt from the INPUT statement by simply using OPEN #3,12,0,"E:" at the beginning of the program and then replacing INPUT with INPUT#3.

Sigh. How right you are, Howard. The funny thing is that I remember discovering this technique about three years ago on our Apple II version of OSS BASIC. How soon we forget. I will incorporate his suggestion in the finished version of BAIT.

Also, Howard protested my not including a facility to list BAIT programs to disk and retrieve them. Perhaps I might change my mind later, but for now I feel that adding that code is an excellent exercise for the reader.

The second letter was from Donald Biresch of Ottsville, Pennsylvania. His comment was that he wished I wouldn't "spend [my] time ... writing about creating BASIC interpreters (something ... less than 1 percent of the end user market has any interest in)." Is he right or wrong? Wrong, I hope, though I admit I have sometimes regretted starting this project, since it has proven to be a larger program than envisioned.

Still, I believe that the subject interests more than 1 percent of you, even if my readers aren't necessarily typical "end users." In particular, I think the BAIT articles are a good lead-in to a more serious study of a BASIC interpreter.

However, if Donald is correct, I apologize. Let me know how you feel.

New Features Of BAIT

As with the previous parts, I will describe this month's changes by line number or line number range.

1110. We set all variables to zero.

1515 to 1580. These are simply some line number equates for use as the objects of GOTOs or GOSUBs. Note, though, that they help produce readable code.

3060. Just centralizing some error messages.

4200 to 4250. A complete restructuring of the "Execute Next Statement" routines. Note that multiple statements per line are now legal. Also, note that pushing the START button now serves as a program break (the BREAK key still stops BASIC itself).

4610 to 4620. Sometimes when you generalize things, the program gets simpler. Direct and deferred execution are now virtually identical.

4700 to 4730. After executing a direct line, we wipe it out of the program memory.

4910 to 4960. Look at all the wonderful statements we can now use! They are in order here. Thus a statement "A" will cause DO ACCEPT to be called, etc.

8290. More clean up.

8400 to 8410. Ditto. See line 3060.

10190. Now, we exit from the statement "DO" routines only after getting the character which terminates the statement (that is, the colon or return character).

10250 to 10270. Ditto. Just making PRINT's code cleaner.

10400 to 10420. Look how easy BEGIN (same as BASIC's RUN) is! We zero out the variables, set the current line number to zero, say we found an end of line, and let execute-next-line (at 4600) start the program execution.

10500 to 10530. GOTO is almost as simple. Find what line number the user wants and fool execute-next-line into getting the next execution line from there.

10600 to 10650. LET is only a little more complex. It insists on a variable (10610) for a destination (10620), an equal sign (10630), and an expression (10640). Then it simply gives the destination variable the value of the expression.

10700 to 10730. IF is, I think, a little clever. It simply tells the get-next-statement code (4240 and 4250) that the next character is an end of line if the user's expression evaluates to zero. Otherwise, it does nothing, and the next statement (if any) gets executed.

10800 to 10910. ACCEPT and CALL will be implemented next month.

11000 to 11030. END simply forces an end of line character and an illegal next line number value. The direct statement test (line 4620) effectively ends the program.

11100 to 11410. FETCH, NEW, RETURN, and STORE are left for next month.

Well, there you have it. A functional, albeit minimal interpreter. If you have typed it all in properly, you might try the following program as a test of its logic.

```
1 P "N",: P "N+N",: P "N*N"
2 P N,: P N+N,: P N*N
3 L N=N+1
4 I N<20: G 2
5 E
B
```

And, for those of you who have not followed BAIT up until now, that translates roughly into BASIC as:

```
1 PRINT "N","N+N","N*N"
2 PRINT N,N+N,N*N
3 LET N=N+1
4 IF N<20 THEN GOTO 2
5 END
RUN
```

And that's enough BAIT for this month. If you don't do anything else while waiting for next month's column, you might try writing the code to execute NEW. It will be *extremely* simple.

BAIT

```
1110 FOR ALPHA=0 TO 26:VARIABLES(ALPHA)=
0:NEXT ALPHA
1515 DIRECT=4700:BADLINE=8400
1560 DOBEGIN=10400:DOGOTO=10500:DOLET=10
600:DOIF=10700
1570 DOACCEPT=10800:DOCALL=10900:DOEND=1
1000:DOFETCH=11100
1580 DONEW=11200:DORETURN=11300:LET DOST
ORE=11400
3060 GOTO BADLINE
```

```
<<< DELETE LINE 3070 >>>
4200 REM EXECUTE A SINGLE STATEMENT
4230 IF PEEK(53279)<>7 THEN GOSUB DOEND
4240 IF C$=":" THEN 4200
4250 IF C>=0 THEN GOTO SYNTAX
4610 CURLINE=CURLINE+1
4620 IF CURLINE>0 AND CURLINE<=MAXLINE T
HEN 4000
<<< DELETE LINE 10280 >>>
4700 REM ===COME HERE ON END OF DIRECT L
INE EXECUTE===
4710 IF LINES(0) THEN BUFFER$(INT(LINES(
0)/1000))="*"
4720 LINES(0)=0
4730 GOTO PROMPT
4910 ERR$="BAD STATEMENT NAME"
4920 ON ALPHA GOTO DOACCEPT,DOBEGIN,DOCA
LL,DODISPLAY,DOEND
4930 ON ALPHA-5 GOTO DOFETCH,DOGOTO,ERRO
R,DOIF,ERROR,ERROR
4940 ON ALPHA-11 GOTO DOLET,ERROR,DONEW,
ERROR,DOPRINT
4950 ON ALPHA-16 GOTO ERROR,DORETURN,DOS
TORE
4960 GOTO ERROR
8290 GOTO DIRECT
8400 REM BAD LINE NUMBER
8410 ERR$="BAD LINE NUMBER":GOTO 8200
10190 GOTO GETNC
10250 IF C$=":" THEN GOTO GETNC
10260 IF C$="," THEN PRINT,:GOTO GETNC
10270 PRINT:RETURN
<<< DELETE 4630 >>>
<<< DELETE 4640 >>>
10400 REM ===EXECUTE BEGIN===
10410 FOR ALPHA=0 TO 26:VARIABLES(ALPHA)
=0:NEXT ALPHA
10420 CURLINE=0:C=-1:RETURN
10500 REM ===EXECUTE GOTO===
10510 GOSUB EXEXP
10520 IF LINES(EVAL)=0 THEN ERR$="NO SUCH
LINE":GOTO 8200
10530 CURLINE=EVAL-1:RETURN
10600 REM ===EXECUTE LET===
10610 GOSUB GETNC:IF NOT ALPHA THEN GOTO
SYNTAX
10620 DESTVAR=ALPHA
10630 GOSUB GETNC:IF C$<>=" " THEN GOTO S
YNTAX
10640 GOSUB EXEXP:VARIABLES(DESTVAR)=EVA
L
10650 RETURN
10700 REM ===EXECUTE IF===
10710 GOSUB EXEXP
10720 IF NOT EVAL THEN C=-1:C$=""
10730 RETURN
10800 REM ===EXECUTE ACCEPT===
10900 REM ===EXECUTE CALL===
10910 GOTO ERROR
11000 REM ===EXECUTE END===
11010 PRINT"===END AT LINE";CURLINE;"===
"
11020 C=-1:CURLINE=C:C$=""
11030 RETURN
11100 REM ===EXECUTE FETCH===
11200 REM ===EXECUTE NEW===
11300 REM ===EXECUTE RETURN===
11400 REM EXECUTE STORE===
11410 GOTO ERROR
```


INSIGHT: Atari

Bill Wilkinson

A mini-series on relocatable machine language begins in this month's column, plus a tip on a new product – an intelligent cable. Next month, the last part of the BAIT interpreter and more on relocatable machine language.

I have been working on a new project for **COMPUTE! Books**. By the time you read this, *COMPUTE!'s Atari BASIC Sourcebook* should be wending its way to your dealers' shelves and into your hands. Like *Inside Atari DOS*, the *Sourcebook* is a complete source listing of – what else? – Atari BASIC, along with a comprehensive explanation of how and why it all works.

Enough advertising. This month we will begin a mini-series on self-relocatable machine language. But before we begin all that, time out for some ruminations.

Machine Language Be Not Hard

Before we start investigating self-relocatable machine code on the 6502, I'd like to get up on my soapbox for a while and do a little preaching.

This month's sermon was inspired by a machine language program published in another magazine. The program seemed to me the epitome of poor programming techniques. And lest it seem that I am taking a cheap shot, let me hasten to add that the program works and works well. I am carping about the printed form of the program, not the results thereof.

In the tradition of any good preacher, then, let me give you some suggestions on how to write good, readable, maintainable machine language:

1. Always use plenty of comments (they cost nothing in the assembled code, unlike BASIC).
2. Never use absolute addresses (except in equates).
3. Never use absolute numeric constants (again, except in equates, though we might forgive an occasional constant 0 or 1).
4. Always use plenty of comments.
5. Always use long, meaningful names for labels. (Which makes more sense, ICCOM or IOCB.COMMAND ?).
6. Never branch to a location relative to the

location counter (that is, never use `"* + xx"` or `"*-xx"`).

7. Never use a comment that simply echoes the machine language code.

8. Always use plenty of comments.

9. Never change the location counter needlessly (that is, most programs should contain only one `"* = "`, except for the use of `"* = * + xx"` to reserve space).

10. If possible, always define a label before its first use.

11. Always thoroughly document the entry and exit values for a subroutine, taking special care to note what happens to the CPU registers.

12. Always use plenty of comments.

Those of you with some OSS software will see that I have taken a small pot shot at our own manuals in commandment 5. Well, I never said we were perfect. (Great, maybe, but not perfect.)

And those of you with Atari's Macro Assembler may object to using long labels since, even though AMAC allows long labels, it ignores all but the first six characters. Sorry, but I still think this rule should be followed. You just have to be more inventive to insure that labels are unique in the first six characters. (For example, IOCB.AUX1 and IOCB.AUX2 look the same to AMAC, so use IOCB.1AUX and IOCB.2AUX.)

Anyway, rather than go through each of those commandments one by one, let's look at an example subroutine coded with both worst and best techniques.

Example 1: Worst Technique

```
; EXAMPLE 1 : print A register
* = $1F00
LDX #11
STX $342 ; put 11 in location $342
LDX #0
STX $348
STX $349
JMP $E456 ; go to $E456
```

Example 2: Best Technique

```
;
; Example 2: Output the character in the A-register
; to file channel (IOCB) number zero
; (assumed to be the screen).
```

; Entry: A-register contains the character
 ; Exit: Status of all registers unknown
 ;

```
*= LOWMEMORY
PRINTCHARACTER
LDX #COMMAND.PUTBINARY
STX IOCB.COMMAND ; command for CIO
LDX #0 ; use a zero buffer length
STX IOCB.LOLENGTH ;tells CIO to output
STX IOCB.HILENGTH ;contents of A register
; next line commented out...not needed since X
; already =0
; LDX #0 ; specify IOCB zero
JSR CIO ; let CIO do the real work
; Could check for errors here
RTS ; all done
```

Enough said? I refuse to decipher programs like Example 1. Of course, Example 2 wouldn't be very useful either unless equates for the various labels were supplied (as in IOCB.COMMAND=\$342), but at least most readers could understand its intent.

Absolutely Not

Regular readers will no doubt recall the many occasions on which I have ranted about staying out of Page 6 or about putting code at LOMEM or about writing code that is not specific to a particular hardware/software configuration. But, to be fair, sometimes it is hard to follow all of the rules, especially when adapting a program from a book or magazine.

Often, the real secret to writing adaptable code is in learning to write self-relocatable code. The techniques we will begin discussing this month are designed specifically for use with the 6502 microprocessor. While there will be several references to Atari internal structure, most of what is presented here is appropriate to Apple and Commodore machines as well.

And I will answer one more question before we start on the hard stuff: *Why* should we want to write self-relocatable code? Sorry, we don't have room for that answer this month. Wait until next month. (It's a good answer, honest!)

Actually, there is just one rule to remember in writing self-relocatable code: *avoid references to absolute memory locations.*

Unfortunately, this is often a very hard rule to follow. Fortunately, there are many places where we can make an exception to this rule.

For starters, look at the subroutine in Examples 1 and 2 above. Is it self-relocatable? Your first impulse might be to say *no*, since it references \$342, \$348, \$349, and \$E456, which are all absolute locations. And even if you do it right and use the equated labels of Example 2, they are still absolute, no matter what they look like.

But. Within the context of any given machine, there are always certain locations which *never* change. In particular, hardware locations, loca-

tions in ROM, and locations in the RAM (or values used and defined by ROM subroutines) cannot possibly change. An exception to this is when you plug in a new set of ROMs, and you can ask the software vendors about the fun and games the Atari 1200XL's new ROMs are giving them.

In the example given, \$E456 (CIO) is in the Atari's OS ROM space. It is a guaranteed entry point to the OS command implementation code. It won't change (even in the new 1200, etc.).

And locations \$340 through \$34F (as well as \$350 through \$3BF) are in the IOCB space defined by Atari for use with CIO. Again, they won't and cannot change.

Finally, the command used (11) and the zero buffer length are values defined by the OS ROMs to have certain meanings. And if Atari changes these meanings, we are *all* in trouble, because Atari BASIC, PILOT, and more won't work then.

Implicit Relocatability

The result of all this? No matter where you assemble that example (that is, no matter where the ***=* places the code), the resultant machine object code will be precisely the same! Presto. That example is self-relocatable.

Surprisingly, a lot of the subroutines used with Atari BASIC follow the mold shown here: they simply set up some values in the Atari-specified memory locations and call an Atari-specified OS routine. They are implicitly self-relocatable.

So what is *not* relocatable? Generally, the prime culprits are:

1. References to RAM locations defined within the user's own code (for example, LDA, STA, INC, etc.).
2. Jumps (JMPs) to locations in the user's own code.
3. Calls (JSRs) to locations in the user's own code.

Let's make up an example just to illustrate potential problems.

```
*= $600
SAVEX *= *+1
MESSAGE .BYTE 'This is the message',0
;
; this is the same code as the examples above
;
PRINTCHARACTER
LDX #COMMAND.PUTBINARY
STX IOCB.COMMAND ; command for CIO
LDX #0 ; use a zero buffer length
STX IOCB.LOLENGTH ;tells CIO to output
STX IOCB.HILENGTH ; contents of A register
JMP CIO ; let CIO do the real work
```

```
;
; call here to print contents of 'MESSAGE'
; Entry conditions: none
; Exit conditions: none, no registers saved
;
```

PRINTMESSAGE

```

LDX #0
STX SAVEX ; initialize message pointer
MSGLOOP
LDX SAVEX ; get current message pointer
LDA MESSAGE,X ; get next character of msg
BEQ QUIT ; but quit if it's last char
JSR PRINTCHARACTER ; else print it
INC SAVEX ; point to next character
JMP MSGLOOP ; and do another character
;
QUIT
RTS ; we are done

```

Do you see the problem areas? If we move this routine somewhere else in memory, the addresses of MESSAGE, PRINTCHARACTER, MSGLOOP, and SAVEX all change, and the object code associated with them changes also. This routine is definitely *not* self-relocatable.

But let's tackle each of the problem labels one at a time and see how we can change the references to each to make the code self-relocatable.

MSGLOOP is the easiest label to "fix." For example, if we change the line JMP MSGLOOP to BNE MSGLOOP, the label MSGLOOP is no longer a problem (since *all* branch instructions are always, by nature, self-relocatable).

And we could save the X-register on the stack (via TXA and PHA) and later retrieve and increment it similarly (via PLA, TAX, and INX), thus eliminating the need for SAVEX.

The PRINTCHARACTER routine could easily be eliminated in its entirety by placing its code in-line in the middle of the PRINTMESSAGE routine. This is a good solution only if PRINTCHARACTER is not called by any other routine. It may also be an adequate solution if the routine being placed in-line is fairly small (as is PRINTCHARACTER) so that you can keep two or more copies around, if necessary.

But what do we do about MESSAGE, which is too big to put in a register? Or what would we do if PRINTCHARACTER was a long routine? And, most importantly, what do we do with a hunk of self-relocatable code once we have managed to produce it?

Next month we'll tackle those questions and others.

A Handy Product?

Do you do much work on *both* Apple II and Atari computers? If so, you could probably use a handy-dandy little device which we recently acquired.

Allen Prowell of Fresno, California, built us what amounts to an intelligent cable between our Apple and our Atari. It plugs into the joystick port on the Atari and into the game port on the Apple. It transfers ASCII files in either direction (doing "light conversion" on return characters, etc.). *Very fast*. It is much more convenient and reliable than using RS-232, and it moves over 1000

characters a second, including disk accesses.

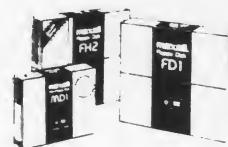
As I said, this is a specialized product, but if you need it, call Allen (209) 227-4917. Using our C/65 and MAC/65 on both Atari and Apple, we have converted an 8K program in as little as two hours, including the transfers, assemblies, etc.

Coming Attractions

I think next month's column will be fairly long, what with the last part of BAIT and Part 2 of self-relocatable machine language. If I have room, though, I will introduce you to a new Atari graphics mode. Also, coming soon, information on some strange and wonderful new products for the Atari.

Maxell Floppy Disks

The Mini-Disks
with maximum
quality.



Dealer inquiries invited. C.O.D.'s accepted.
Call FREE (800) 235-4137.



PACIFIC EXCHANGES
100 Foothill Blvd
San Luis Obispo, CA 93401
In Cal call (800) 592-5945 or
(805) 543-1037



ENHANCE YOUR ATARI 810

HAPPY 810 ENHANCEMENT

Speed up program development, loading, execution, and copying time by reading disks up to 3 times faster. Complete compatibility with existing software, with faster disk initialization, and reduced wear on the disk drive mechanism. No soldering or trace cutting required, complete installation instructions included, or contact your dealer. Diagnostic program included.

SOFTWARE ENHANCEMENTS (require HAPPY 810 ENHANCEMENT)

HAPPY BACKUP PROGRAM

Guaranteed to produce executable backup copies of any disk which can be read with a standard ATARI 810* disk drive. Backup those important disks in your library or use HAPPY BACKUP for small scale software production. Completely automatic duplication of format and data content of the source disk. Single and multiple drive versions available. Backup copies will work on a drive without the enhancement.

HAPPY COMPACTOR PROGRAM

Combines self booting programs which reside one per disk into one disk with many self booting programs using the HAPPY COMPACTOR file structure. Programs are then executed from the self booting HAPPY COMPACTOR menu, and may later be extracted back onto a single disk. Compacted programs disk will execute only on a drive which has the HAPPY 810 ENHANCEMENT. Pays for itself by reducing the number of backup disks you need, in addition to the added convenience.

HAPPY CUSTOMIZER PROGRAM

User friendly program to generate source disks with custom track format. Format is specified on a per track basis. Examples of usage and interpretation of results are included. This system requires a more advanced level user.

HAPPY 810 ENHANCEMENT WITH SINGLE DRIVE HAPPY BACKUP	\$249.95
MULTIPLE DRIVE HAPPY BACKUP PROGRAM	\$ 49.95
HAPPY COMPACTOR PROGRAM	\$ 49.95
HAPPY CUSTOMIZER PROGRAM	\$ 99.95

CALL OR WRITE FOR ORDERING INFORMATION. Sorry, no COD or credit cards accepted. Dealers may inquire, send letterhead.

HAPPY COMPUTING
P.O. Box 32331
San Jose, CA 95152
(408) 251-6603



*ATARI 810 is a trademark of ATARI Inc.

INSIGHT: Atari

Bill Wilkinson

I've been a bit remiss about my column recently. The editorial staff at **COMPUTE!** has covered nicely for me, splitting some of my larger articles into two parts and cutting and pasting. I shall try to make life easier for them for the next few months, since I have finally accumulated a mental backlog of material which I feel is suitable for this column.

Mind you, I can still use some input from you readers on what you would like to see, so don't stop writing. As I have stated often in the past, it doesn't seem ethical for me to review software; but that shouldn't keep me from commenting on books, hardware, and who knows what else.

And, in that vein, this isn't truly a "review," since I have not had a chance to actually try it yet, but the most interesting new product for the "serious" Atari owner that I have seen lately is the new 64K byte memory card from Mosaic Electronics. With it you can make your 800 behave just like a 1200 so far as the bank selecting of RAM versus ROM goes. Mosaic rightly points out that there is zero software currently available to take advantage of the RAM which must lie where the OS ROMs are, so perhaps the other configuration of their RAM board makes more sense. How about up to 192K bytes of RAM in an Atari 800, with all but the first 48K being bank selected in 4K hunks that reside at \$C000 through \$CFFF. That gives you 36 little 4K byte banks, so just imagine the graphics switching you might do (in modes 7 and below only, though)! It's not cheap, but it certainly seems like a solution looking for a problem.

Predictions Revisited

I was right on two counts! First, I said the 1200 was overpriced. But look at the prices now. I am seriously considering buying one. Or I *was*. Because I just heard that Atari is *dropping* the 1200! Welcome, welcome, Atari 600, 1400, and 1450, which were introduced at the Summer Consumer Electronics Show. All will have expansion capability like nothing Atari has built before. So watch out world: here come the add-ons. [For more on the new Atari products at CES, see Tom Halfhill's article "The Fall Computer Collection At The Summer Consumer Electronics Show" elsewhere in this issue.]

Since, by the time you read this, the announcements will have been made, you will be able to see how good my rumor sources and crystal

ball gazers are. Me? I'm sitting on the edge of my chair for another week or two.

One more thing before we get to the meat of this month's column. It would appear that I fooled more than a few people with my April column. If you were fooled, I apologize. But not much. After all, April Fool articles in computer magazines are a tradition that goes back to the first days of *Data-mation* (a magazine sent free to anyone who owns a computer worth more than a quarter million dollars, heavily loaded with IBM mainframe articles, but it wasn't always so). Be assured that if you were fooled you were in good company: I showed the article to a COBOL programmer with ten years experience, and she didn't get it either. (To be fair to me, though, didn't you notice the title of the column that month, "Outasight: Atari"?)

Well, enough chitchat. Shall we tackle BAIT one more time? I am not sorry to see this series end, but looking at the finished product I can honestly say that those who understand it (and know at least a smattering of machine language) should be able to tackle **COMPUTE!'s** *Atari BASIC Sourcebook*, wherein we detail the workings of a *real* interpreter.

BAIT, Part 4

This month we present the listing of BAIT in its entirety. It is not a small listing, and there is no room in a single column to recap all the details of its creation and function. So, you really need Parts 1 through 3 (which appeared in March, May, and June) if you want the full design principles.

As a very brief summary, though, let's mention the following:

1. BAIT is a very simple pseudo-BASIC interpreter which has been written in Atari BASIC.
2. BAIT accepts only single-letter statement names (as shown in the table) and single-letter variable names (A through Z).
3. BAIT allows BASIC-style screen editing, line numbering, etc., with the restriction that line numbers must be from 1 to 99.
4. There is no precedence of operators, parentheses, functions, or any other amenities. This is a *primitive* language.

Does it work? Yes. Is it useful? Only as a learning tool. Could it be made useful? If we wrote a compiler for the same language, maybe.

New Goodies

This month, I have finally implemented the rest of the statements listed in the table. In particular, we now have Accept, Call, Fetch, New, Return, and Store available to us.

New and Return function exactly like their BASIC counterparts of the same names. Accept, Call, and Store are simply different names for BASIC's INPUT, GOSUB, and POKE, respectively. They had to be named as they are to implement the single-letter statement names.

Fetch, then, is the only strange statement. It owes its existence to the fact that BAIT doesn't allow functions. Generally, Fetch is equivalent to PEEK, but its format is that of POKE (and, naturally, Store). It does, however, require a variable to store its Fetched value in (much like GET in Atari BASIC).

The statements are fairly straightforward, and we shall see more of them a little later on. For now, though, let's analyze the additions and changes made to BAIT this month on a line-by-line basis. The lines discussed below are those which have changed or been added since the June column. If you have typed in BAIT as we have proceeded through parts 1, 2, and 3, you may enter just those lines.

- **Line 1130.** This is the stack we will use for "remembering" where Calls (GOSUBs) were made from. The size is arbitrary, but I cheated and used a fixed number, so don't change it unless you also change line 10910.

- **1720.** This makes screen editing of BAIT programs very, very much easier. See line 2300.

- **2200.** We always reset the Call stack pointer because program editing could invalidate any or all pending Return locations.

- **2300.** See line 1720. This is how we eliminate the "?" prompt from the screen when using the INPUT statement. A clever trick: use it in all your programs. It comes to you courtesy of Howard Fishman. Thanks, Howard.

- **2360.** Notice that this line (which used to strip off the question mark) is now gone. You won't miss it.

- **1540, 5520, and 5530.** The TRAP to BAD-VALUE was added just in case your BAIT program generated an overflow.

- **8310 and 8410.** Cosmetic changes only.

- **8500 and 8510.** A new error message. It's used for all BAIT numeric data problems.

- **10210.** A minor change to allow Print (without a following expression) to be followed by a colon statement separator.

- **10530-10550.** A fix. Without it, the Goto doesn't occur until the end of the line. Thus 'G 10

: P "oops" ' would indeed print the "oops" until now. But this fixes it.

- **10810-10860.** Finally, some new code! Actually, Accept is fairly simple and closely follows the format of Let. Instead of requiring an expression after an equals sign, though, Accept wants the user to INPUT something from the keyboard. Thanks to the TRAP, only numeric data will be allowed.

- **10910-10960.** We process the Call statement. Line 10910 seems unnecessary: who would *want* to go 50 levels deep in a BAIT program? But it works. Notice that all three vital pointers must be saved on the stack. Could it have been done more compactly? Yes, but this way is much simpler. Finally, we allow Goto to do the real work of transferring control to a new line number.

- **11110-11150.** Fetch also follows the form of Let, but in reverse. First we get an address (line 11110), then a comma (line 11120), and finally a variable to put the Fetched value into (line 11130). The TRAP of line 11140 insures that the address given was a legal one.

- **11310-11370.** Return is the opposite of Call. Again, line 11310 is for safety only; good programmers can't make mistakes like this, right? Lines 11320 to 11350 restore the information saved by Call in lines 10920 to 10950. Finally, since we saved CURLOC before we joined the Goto processing, we must skip over the line number expression to find out if there is a colon (":") waiting for us.

- **11410-11450.** Store is almost identical to Fetch. The exception: the item after the comma can be any expression at all; it does not need to be a simple variable. Again, the TRAP in line 11440 insures against illegal addresses and/or data.

Sampling The BAIT

Well, we can presume that you typed all of BAIT in properly, yes? So let's quickly try some BAIT programs, to see what you can do in the language.

Caution: The lowercase letters shown in these listings are there for clarity only! BAIT accepts *only* single-letter commands, so just leave out all lowercase letters. Do *not* convert them to uppercase. For example, the first line of Program 1 should actually be typed in as '1 S 20,0' (and even the spaces may be left out if desired).

Program 1: Tick-Tock

```
1 Store 20,0
2 Print "SHOWING HOW SLOW BAIT IS"
3 Fetch 20,T
4 Print "THAT TOOK ";: Print T;: Print "CLOCK
  TICKS"
5 End
D
B
```

Program 2: Recursion

```
1 Print "GIVE ME AN INTEGER NUMBER";  
  Accept N  
2 Let A=1 : Call 10  
3 Print "THE FACTORIAL OF YOUR NUMBER  
  IS "; : Print A  
4 Print : Print : Goto 1  
10 If N<2 : Return  
11 Let A=A*N : Let N=N-1  
12 Call 10  
13 Return  
D  
B
```

Challenge: Can you modify BAIT so that it will, indeed, ignore the lowercase letters? If so, your BAIT programs could be more readable.

Whew!

And there you have it. BAIT in all its glory. Or is that gory? Some carpers may claim that the only thing it proves is that people will try to write *anything* in BASIC. I like to think it may have provided a way for some of you to understand the mechanics of an interpreter. If it helps turn even one or two people into systems-level programmers, it will have done its job.

But if BAIT didn't interest you, don't worry. There are even a few out there that don't like to program games. (I certainly like to play them. I'm hooked on - oops, can't review software here, sorry.)

Self-relocatable Machine Language, Part 2

Last time we were on this subject, I promised to give a reason why we would want to write self-relocatable machine language. And sometimes I even keep my promises.

The primary advantage of self-relocatable code is, obviously, that you can load it and run it anywhere in memory. But why would you want to do that? Why not just decide where the code will go and leave it at that? Well, let's try to answer those questions.

First of all, none of what I am about to say pertains to programs which "take over" the system. After all, if you *know* that your code will run in such and such a way because, for example, you only give it out on a heavily protected game disk, then you can obviously place various hunks of machine language exactly where you want them. And they'll stay put.

But a large proportion of my readers are, I believe, attempting to either write machine language programs which interface to BASIC or are attempting to add on to the operating system in some way. In both these instances, self-relocatable code is invaluable.

Why? Because there simply isn't very much room in the Atari memory map that isn't used for

something or other. In point of fact, the only clear portion of memory seems to be the infamous "Page Six." But, remember, even Atari BASIC can clobber the lower half of that page. And BASIC A+, Microsoft BASIC, Atari PASCAL, and several other products use portions or all of Page Six. What to do?

Well, if you have been following my articles, you will know that I advocate placing your program at LOMEM, moving LOMEM up to cover your program, and hooking into the system reset chain so that you can preserve your program if the user hits the reset key.

All well and good, but suppose LOMEM moves? And it will and it does. Depending on the number of disk drives and/or files you need to support, LOMEM can be anywhere from \$A20 (with OSS PicoDOS) to \$1D00 (standard Atari DOS) to \$2C00 (OS/A+ version 4.1). And, if the RS-232 drivers are to be loaded (for the 850 interface), you can count on LOMEM being even higher still.

What's a poor old machine language programmer supposed to do? Follow my directions, natch. Put your program at LOMEM, no matter where it is. And that's easy to do if your program is self-relocatable.

And, before we get into discussing *how* to write this magic kind of program, I would like to point out one other significant instance where self-relocatable programs are handy. Putting programs at LOMEM and moving LOMEM up is all very well and good if you can do that before BASIC gets control. But once the language is entered, it has already noted the contents of LOMEM and used them for its own initialization purposes. Changing LOMEM will not necessarily force BASIC to move its own internal LOMEM, and you may wind up with a conflict of usage.

But there is a hunk of memory which is properly handled by BASIC as far as we are concerned: strings. Any data, including a machine language program, placed in a dimensioned string is guaranteed to be moved around intact (for example, when a new program line is entered or when a new variable is introduced).

Indeed, there have been many articles published which put a machine language routine or two in a string and then call the routine via `USR(ADR(strings$),...)`. In fact, I have even seen a few adventuresome souls who have used `ADR("some graphics and other characters here")`. That is, it is perfectly O.K. to take the address of a literal string, also.

For the rest of this series, I will presume that we are writing programs which are designed to reside in Atari BASIC strings. I think that is sufficient, since there is little, if any, difference in con-

cept between placing programs in strings and placing them at a potentially movable LOMEM.

From Why To How

Let's begin by listing the things you *don't* have to worry about when writing self-relocatable programs. Some of these things were discussed briefly last month; others are new but should be fairly obvious. The following, then, are intrinsically "safe" types of machine language:

1. All instructions which involve only one or more registers (e.g., TAX, PHA, INY, etc.).
2. All load immediate instructions which do *not* involve the address of a location as the immediate value (e.g., LDA #5, but not LDY #LOCATION/256).
3. All branch instructions (BNE, BCC, etc.).
4. All instructions involving *fixed* operating system or language specific locations, either in ROM or RAM (e.g., STA LEFTMARGIN, JSR CIO).
5. Several miscellaneous instructions which do not reference memory addresses, such as SED, SEI, CLC, NOP, RTS, etc.

What about the intrinsically unsafe instructions? Here is one of them:

Any instruction which references an absolute memory location within your own code (or another block of relocatable code) or which references a fixed RAM location which is not dedicated to the purpose intended.

Now, that's not so bad. There are a lot more safe conditions than unsafe ones, aren't there? And, yet, it takes only one unsafe instruction to clobber you, so let's concentrate on some techniques for avoiding the unsafe conditions.

Safe Relocatable Techniques

1. Change JMPs to branches. Usually, you can do a CLC followed by a BCC to substitute for a JMP. Sometimes, the target of the jump is too far away, though. In that case, add an intermediate branch point, so that the first BCC branches to a second BCC, etc.

2. Save register values on the stack (via TAX, PHA, etc.) rather than in fixed RAM locations. If you need to save a value in between calls from a higher level routine (e.g., the BASIC program), though, you will *have* to find some safe place to put it. Watch out! There are only four safe locations in zero page and only a handful in other parts of memory. More about such safe locations in the next part in this series.

3. If you need to reference bytes in a table, string, or other portion of memory, why not let BASIC handle the addressing for you? For example, consider this BASIC line:

```
TEST = USR(ADR(CODE$), ADR(TABLE$))
```

Presuming that your machine language routine is in CODE\$, it can then reference TABLE\$ as follows:

```
PLA                ;parameter count
PLA
STA ZTEMP+1        ;high byte of address
PLA
STA ZTEMP          ;low byte of address
LDY #0
LDA (ZTEMP),Y      ;get first byte of the table ...
```

That program fragment is certainly intrinsically relocatable (except for the location of ZTEMP, but it needn't be preserved in between calls to the fragment). And BASIC will certainly move TABLE\$ around as it needs, giving you the address when you need it.

4. If you absolutely *have* to use a hunk of nonrelocatable programming, and you don't have space to keep it on a permanent basis, why not temporarily move it from a relocatable location (e.g., TABLE\$ in our example above) to a fixed location (e.g., BASIC's input buffer at \$580 or some such). Then you can use it safely there, without worrying about relocatability. Of course, each time you are called from BASIC you would have to move the routine. But, as slow as BASIC is, you might never notice the extra overhead.

Next time we will continue right here. We will try to develop some even more useful techniques, including one which can only be used with USR calls from BASIC. Stay tuned.

BAIT Statements

A Accept <variable>	(INPUT)
B Begin	(RUN)
C Call <line-number>	(GOSUB)
D Display	(LIST)
E End	
F Fetch <address>, <variable>	(pseudo-PEEK)
G Goto <line-number>	
I If <expression>, <statement>	
L Let <variable> = <expression>	
N New	
P Print <string-literal>	
Print <variable>	
Print	
R Return	
S Store <address>, <expression>	(POKE)

BAIT

```
1000 REM ..INITIALIZATION..
1001 REM .....
1010 MAXLINE=99
1020 DIM BUFFER$(5000), LINE$(128)
1030 DIM LINES(MAXLINE)
1040 FOR LP=0 TO MAXLINE: LINES(LP)=0: NEXT LP
1050 BUFFER$=""
1100 DIM C$(1), VARIABLES(26)
1110 FOR ALPHA=0 TO 26: VARIABLES(ALPHA)=0: NEXT ALPHA
```

```

1120 DIM ERR$(40)
1130 DIM STACK(50,2):REM MAX CALLS THUS
    IS 50
1500 REM LINE NUMBERS OF EXECUTION ROUTI
    NES
1510 PROMPT=2100:INNEX=2300
1515 DIRECT=4700:BADLINE=8400
1520 LET GETNC=8100
1530 SYNTAX=8300:ERROR=8200:EXEXP=5000
1540 BADVALUE=8500
1550 DODISPLAY=10100:DOPRINT=10200
1560 DOBEGIN=10400:DOGOTO=10500:DOLET=10
    600:DOIF=10700
1570 DOACCEPT=10800:DOCALL=10900:DOEND=1
    1000:DOFETCH=11100
1580 DONEW=11200:DOReturn=11300:LET DOST
    ORE=11400
1700 REM MISCELLANY
1710 UNTRAP=40000
1720 OPEN #5,12,0,"E.":REM SO THERE IS N
    O ? PROMPT
2000 REM ..INTERACTION..
2001 REM .....
2100 PRINT "READY"
2200 STACK=0:REM CLEAN UP 'CALL' STACK
2300 INPUT #5,LIN$
2350 IF LEN(LIN$)=0 THEN GOTO INNEX
    <<< DELETED OLD LINE 2360 >>>
2370 LL=LEN(LIN$)
2500 REM CHECK FOR LINE NUMBER
2510 FOR LP=1 TO LL
2520 IF LIN$(LP,LP)<="9" AND LIN$(LP,L
    P)>="0" THEN NEXT LP
2550 REM LP HAS POSITION OF FIRST NON-NU
    MERIC CHARACTER
2560 CURLINE=0
2570 IF LP>1 THEN CURLINE=VAL(LIN$(1,LP
    -1))
2600 REM NOW SKIP LEADING SPACES, IF ANY
2610 IF LP>LL THEN 2700
2620 FOR LP=LP TO LL
2630 IF LIN$(LP,LP)=" " THEN NEXT LP
2700 REM REMOVE LINE NUMBER AND LEADING
    SPACES
2710 IF LP>LL THEN LIN$="":GOTO 3000
2720 LIN$=LIN$(LP)
3000 REM ..EDITING..
3001 REM .....
3010 REM IF HERE, LINE NUMBER IS IN CURL
    INE
3020 LL=LEN(LIN$):REM AND LL IS LENGTH
    THEREOF
3030 IF CURLINE=0 AND LL=0 THEN GOTO PRO
    MPT
3040 IF CURLINE<>INT(CURLINE) THEN 3060
3050 IF CURLINE<=MAXLINE THEN 3100
3060 GOTO BADLINE
3100 REM FIRST, DELETE CURLINE IF IT ALR
    EADY EXISTS
3110 LENGTH=LINES(CURLINE):IF LENGTH=0 T
    HEN 3200
3120 START=INT(LENGTH/1000)
3130 LENGTH=LENGTH-1000*START
3140 BUFFER$(START)=BUFFER$(START+LENGTH
    )
3150 LINES(CURLINE)=0
3160 FOR LP=1 TO MAXLINE:TEMP=LINES(LP)
3170 IF TEMP>=START*1000 THEN LINES(LP)=
    TEMP-LENGTH*1000
3180 NEXT LP

```

```

3200 REM NOW ADD LINE TO END OF BUFFER
3210 IF LL=0 THEN GOTO INNEX
3220 START=LEN(BUFFER$)
3230 BUFFER$(START)=LINES$
3240 BUFFER$(LEN(BUFFER$)+1)="*"
3250 LINES(CURLINE)=START*1000+LL
3300 REM NOW LINE IS IN BUFFER...WHAT DO
    WE DO
3310 IF CURLINE THEN GOTO INNEX
3320 REM NOTE THAT CURLINE=0 AS WE FALL
    TO LINE 4000
4000 REM ..EXECUTE CONTROL..
4001 REM .....
4010 LENGTH=LINES(CURLINE):IF LENGTH=0 T
    HEN 4600
4020 CURLOC=INT(LENGTH/1000):LENGTH=LENG
    TH-1000*CURLOC
4030 CUREND=CURLOC+LENGTH-1
4040 IF CURLINE=0 THEN CURLINE=-1
4100 REM READY TO EXECUTE A LINE
4200 REM EXECUTE A SINGLE STATEMENT
4210 GOSUB GETNC:IF NOT ALPHA THEN GOTO
    SYNTAX
4220 GOSUB 4900
4230 IF PEEK(53279)<>7 THEN GOSUB DOEND
4240 IF C$=":" THEN 4200
4250 IF C>=0 THEN GOTO SYNTAX
4600 REM COME HERE FOR NEXT LINE
4610 CURLINE=CURLINE+1
4620 IF CURLINE>0 AND CURLINE<=MAXLINE T
    HEN 4000
4700 REM ===COME HERE ON END OF DIRECT L
    INE EXECUTE===
4710 IF LINES(0) THEN BUFFER$(INT(LINES(
    0)/1000))="*"
4720 LINES(0)=0
4730 GOTO PROMPT
4900 REM THE STATEMENT CALLER
4910 ERR$="BAD STATEMENT NAME"
4920 ON ALPHA GOTO DOACCEPT,DOBEGIN,DOCA
    LL,DODISPLAY,DOEND
4930 ON ALPHA-5 GOTO DOFETCH,DOGOTO,ERRO
    R,DOIF,ERROR,ERROR
4940 ON ALPHA-11 GOTO DOLET,ERROR,DONEW,
    ERROR,DOPRINT
4950 ON ALPHA-16 GOTO ERROR,DOReturn,DOS
    TORE
4960 GOTO ERROR
5000 REM ..EXECUTE EXPRESSION..
5001 REM .....
5010 EVAL=0:LASTOP=-1
5020 VALID=0
5100 GOSUB GETNC:IF ALPHA THEN 5300
5110 IF C$>="0" AND C$<="9" THEN 5400
5120 REM WHICH OPERATOR?
5121 IF C$="+" THEN OP=1:GOTO 5200
5122 IF C$="-" THEN OP=2:GOTO 5200
5123 IF C$="*" THEN OP=3:GOTO 5200
5124 IF C$="/" THEN OP=4:GOTO 5200
5125 IF C$=">" THEN OP=5:GOTO 5200
5126 IF C$="<" THEN OP=6:GOTO 5200
5127 IF C$="=" THEN OP=7:GOTO 5200
5128 IF C$="#" THEN OP=8:GOTO 5200
5160 IF VALID THEN RETURN
5170 GOTO 5900
5200 REM GOT AN OPERATOR
5210 IF LASTOP>0 THEN 5170
5220 IF LASTOP<0 AND OP>2 THEN 5170
5230 LASTOP=OP:VALID=0:GOTO 5100
5300 REM GOT A VARIABLE

```

```

5310 VAL2=VARIABLES(ALPHA):GOTO 5500
5400 REM GOT A NUMERIC
5410 CURLOC=CURLOC-1:REM BACKUP TO FIRST
    NUMERIC
5420 FOR LL=CURLOC TO CUREND:C$=BUFFER$(
    LL)
5430 IF (C$>="0" AND C$<="9") OR C$="."
    THEN NEXT LL
5440 VAL2=VAL(BUFFER$(CURLOC,LL-1))
5450 CURLOC=LL
5500 REM VAR OR NUMERIC
5510 IF LASTOP=0 OR ABS(LASTOP)>8 THEN 5
    900
5520 TRAP BADVALUE:GOSUB 5600+10*ABS(LAS
    TOP)
5530 TRAP UNTRAP:LASTOP=0:VALID=1:GOTO 5
    100
5600 REM EXECUTE OPERATORS
5610 EVAL=EVAL+VAL2:RETURN
5620 EVAL=EVAL-VAL2:RETURN
5630 EVAL=EVAL*VAL2:RETURN
5640 EVAL=EVAL/VAL2:RETURN
5650 EVAL=(EVAL>VAL2):RETURN
5660 EVAL=(EVAL<VAL2):RETURN
5670 EVAL=(EVAL=VAL2):RETURN
5680 EVAL=(EVAL<>VAL2):RETURN
5900 ERR$="INVALID EXPRESSION":GOTO ERRO
    R
8000 ..MISCELLANEOUS SUBROUTINES..
8001 REM .....
8100 REM GETNC
8110 IF CURLOC>CUREND THEN C=-1:C$=CHR$(
    155):GOTO 8140
8120 C=ASC(BUFFER$(CURLOC)):C$=CHR$(C)
8130 CURLOC=CURLOC+1
8140 IF C=32 THEN GOTO GETNC
8150 ALPHA=(C$>="A" AND C$<="Z")*(C-64)
8160 RETURN
8200 REM ERROR ROUTINE
8210 PRINT :PRINT "****";ERR$;"****";
8220 IF CURLINE>0 THEN PRINT " AT LINE "
    ;CURLINE
8230 PRINT :TRAP 8250
8240 POP :POP :POP :POP :POP :POP :POP :
    POP
8250 TRAP UNTRAP
8290 GOTO DIRECT
8300 REM SYNTAX ERROR
8310 ERR$="SYNTAX ERROR":GOTO ERROR
8400 REM BAD LINE NUMBER
8410 ERR$="BAD LINE NUMBER":GOTO ERROR
8500 REM VALUE OUT OF RANGE ERROR
8510 ERR$="BAD VALUE":GOTO ERROR
10000 REM ..EXECUTE THE VARIOUS STATEMEN
    TS..
10001 REM .....
    ....
10100 REM ==EXECUTE DISPLAY==
10110 FOR LP=1 TO MAXLINE
10120 LENGTH=LINES(LP):IF LENGTH=0 THE 1
    0150
10130 START=INT(LENGTH/1000):LENGTH=LENG
    TH-1000*START
10140 PRINT LP;" ";BUFFER$(START,START+L
    ENGTH-1)
10150 NEXT LP
10190 GOTO GETNC
10200 REM ==EXECUTE PRINT==
10210 GOSUB GETNC:IF C<0 OR C$=":" THEN
    PRINT :RETURN

```

```

10220 IF C=34 THEN 10300
10230 CURLOC=CURLOC-1
10240 GOSUB EXEXP:PRINT EVAL;
10250 IF C$=";" THEN GOTO GETNC
10260 IF C$="," THEN PRINT :GOTO GETNC
10270 PRINT :RETURN
10300 FOR LL=CURLOC TO CUREND:C$=BUFFER$(
    LL)
10310 IF ASC(C$)<>34 THEN PRINT C$;:NEXT
    LL:PRINT :RETURN
10320 CURLOC=LL+1:GOSUB GETNC
10330 GOTO 10250
10400 REM ===EXECUTE BEGIN===
10410 FOR ALPHA=0 TO 26:VARIABLES(ALPHA)
    =0:NEXT ALPHA
10420 CURLINE=0:C=-1:RETURN
10500 REM ===EXECUTE GOTO===
10510 GOSUB EXEXP
10520 IF LINES(EVAL)=0 THEN ERR$="NO SUC
    H LINE":GOTO 8200
10530 CURLINE=EVAL-1
10540 C=-1:C$=""
10550 RETURN
10600 REM ===EXECUTE LET===
10610 GOSUB GETNC:IF NOT ALPHA THEN GOTO
    SYNTAX
10620 DESTVAR=ALPHA
10630 GOSUB GETNC:IF C$<>=" " THEN GOTO S
    YNTAX
10640 GOSUB EXEXP:VARIABLES(DESTVAR)=EVA
    L
10650 RETURN
10700 REM ===EXECUTE IF===
10710 GOSUB EXEXP
10720 IF NOT EVAL THEN C=-1:C$=""
10730 RETURN
10800 REM ===EXECUTE ACCEPT===
10810 GOSUB GETNC:IF NOT ALPHA THEN GOTO
    SYNTAX
10820 TRAP 10850:INPUT EVAL:TRAP UNTRAP
10830 VARIABLES(ALPHA)=EVAL
10840 GOTO GETNC
10850 PRINT "??? MUST INPUT A NUMBER, RE
    PEAT..."
10860 GOTO 10820
10900 REM ===EXECUTE CALL===
10910 IF STACK=50 THEN ERR$="TOO MANY CA
    LLS":GOTO ERROR
10920 STACK(STACK,0)=CURLOC
10930 STACK(STACK,1)=CUREND
10940 STACK(STACK,2)A=CURLINE
10950 STACK=STACK+1
10960 GOTO DOGOTO
11000 REM ===EXECUTE END===
11010 PRINT"===END AT LINE ";CURLINE;"==
    ="
11020 C=-1:CURLINE=C:C$=""
11030 RETURN
11100 REM ===EXECUTE FETCH===
11110 GOSUB EXEXP
11120 IF C$<>"," THEN GOTO SYNTAX
11130 GOSUB GETNC:IF{2 SPACES}NOT ALPHA
    THEN GOTO SYNTAX
11140 TRAP BADVALUE:VARIABLES(ALPHA)=PEE
    K(EVAL)
11150 TRAP UNTRAP:GOTO GETNC
11200 REM ===EXECUTE NEW===
11210 RUN
11300 REM ===EXECUTE RETURN===
11310 IF STACK=0 THEN ERR$="NO MATCHING

```



```

CALL":GOTO ERROR
11320 STACK=STACK-1
11330 CURLOC=STACK(STACK,0)
11340 CUREND=STACK(STACK,1)
11350 CURLINE=STACK(STACK,2)
11360 GOSUB EXEXP:REM IGNORE...ALREADY P
      ROCESSED
11370 RETURN
11400 REM ===EXECUTE STORE===
11410 GOSUB EXEXP:ADDRESS=EVAL
11420 IF C$<>"", THEN GOTO SYNTAX
11430 GOSUB EXEXP
11440 TRAP BADVALUE:POKE ADDRESS,EVAL
11450 TRAP UNTRAP:RETURN

```

Use the card in
the back of this
magazine to order
your
COMPUTE! Books

Get More From Your PET/CBM!

NEW! • 24K MEMORY EXPANSION (\$129-\$239)
Give your PET/CBM a boost to 32K!
Loaded with nifty features. Low, low power.

• **"Real World" SOFTWARE** (\$17 - \$25)
Word Processor, Mailing List, Catalog, Ham Radio, Frequency Counter.
"OLD" 8K PETS

• **2114 - TO - 6550 RAM ADAPTER** (\$12 - \$25)
Replace 6550 RAMs with low cost 2114s. Hundreds Sold!

• **4K MEMORY EXPANSION** (\$16 - \$62)
Low cost memory expansion using 2114s for bigger programs.

Write for FREE Catalog!

OPTIMIZED DATA SYSTEMS
Dept. C, P.O. Box 595 - Placentia, CA 92670

DISK-O-MATE trademark Optimized Data Systems - PET/CBM trademark Commodore

C-64/VIC 20/PET/CBM OWNERS

ROADTOAD - Hop your toad across 5 lanes of traffic, avoid deadly snakes, and dodge the dreaded toad-eaters. Cross a raging river full of logs, turtles, alligators, and park your toad in the safety of a harbor. Each time you park 5 toads, you enter a tougher level where the action is faster and the toad-eaters are more numerous. ROADTOAD is written in machine language and uses high resolution graphics. The sound effects are excellent and you can use a joystick or the keyboard to control your toad.
CASS/5K/VIC 20/C-64 (Includes Shipping/Handling) **\$19.95**
[CALIF. RES. ADD 6% SALES TAX]

CHICKEN CHASE - Help your hapless hen avoid hungry chicken hawks, sneaky coyotes, and fiendish zompy. If your chicken gets into trouble, "hyper-hen" to a new spot on the maze. If your chicken travels the entire maze, you advance to the next level where the action is faster and the predators more numerous. Hi-res graphics, great sounds, and machine language help make CHICKEN CHASE a hilarious fun-filled game for the whole family.
CASS/5K/VIC-20/C-64 (Includes Shipping/Handling) **\$19.95**
[CALIF. RES. ADD 6% SALES TAX]

Write For FREE Catalog **NIBBLES & BITS, INC.** **Write For FREE Catalog**
P.O. BOX 2044
ORCUTT, CA 93455

NEW FOR ATARI diskwiz

**COMPLETE & AFFORDABLE
DISK EDITING REPAIR & DUPLICATION
SYSTEM FOR ATARI OR PERCOM DRIVES**

• single load • fast mach. lang. • repair, explore, dup dos/non-dos sectors • simultaneous hex/ascii display and editing • print out all modes to any printer • dumps special & inverse chars to EPSON grafix & NEC 8023 • fast mapping and byte searches • file link trace • speedcheck and adjust • block move • auto link pointer, file code change • vtoc bit map changes or check • cross sector disassembler • fast/slow copy • 1 or 2 drives • hex-dec-asc conv. • complete manual • create "bad" sectors • fix deleted or open files • fix dup filenames • safely use non-formattable disks • easy, fast, complete • see review Analog 11 • now for 1200XL too!

All this for only \$28.95 postpaid

Don't waste your money on simple copiers or more expensive programs that don't deliver as much.

48 hr. shipping for cashiers checks & money orders. Allow up to 3 weeks for personal checks, — C.O.D. add \$2.00.

Club & dealer enquiry encouraged.

Soon to be released: **PRINTWIZ**

Ask for it at your local dealer.

**ALLEN
MACROWARE** (213) 376-4105
P.O. Box 2205
Redondo Beach, CA 90278
1906 Carnegie Lane "E"

Atari, Epson, NEC & Percom, are trademarks of Atari, Inc., Epson America, Nippon Electric Company, Percom Data respectively.

COMMODORE USERS

**Join the largest, active Commodore users group.
Benefit from:**

- Access to hundreds of public domain programs on tape and disk for your Commodore 64, VIC 20 and PET/CBM.
- Informative monthly club magazine **THE TORPET.**

Send \$1.00 for Program & Information Catalogue.
(Free with membership).

Membership	Canada	—	\$20 Can.
Fees for	U.S.A.	—	\$20 U.S.
12 Months	Overseas	—	\$30 U.S.

Toronto Pet Users Group
Department "S"
1912A Avenue Road, Suite 1
Toronto, Ontario, Canada M5M 4A1

* LET US KNOW WHICH MACHINE YOU USE *

INSIGHT: Atari

Bill Wilkinson

The new 600XL and 1400XL computers were exactly what I expected (except that Atari goofed and changed the number on the 1201 XL – and that's a joke until you study the case designs of the 1200XL and 1400XL). The 800XL was a little bit of a surprise, but kind of a logical step now that I have the benefit of hindsight. The 1450XLD was a pure delight.

I really could envision a 1450XLD doing some nice, small business work. Especially if you put one of the new three-inch hard disk drives (that's over four megabytes of disk space) into that empty space supposedly designed for a second floppy.

If Atari has any problems at all with the XL line of computers, it may be simply that they are priced too close together. After all, an 800XL is essentially a 600XL with 64K of RAM, and the already announced RAM-pack for the 600XL ends up producing an equivalent machine for the same price. Redundancy.

The 1400XL suffers a little, also. After all, if the rumored price of the 1450XLD holds up (\$800-\$900 retail), why would you buy a 1400XL and then add a snail's-pace 1050 drive when you can have the much faster XLD for less money? And who but the more sophisticated user will buy a 1400XL when the 600XL (even with expansion to 64K) is so much less? Will the modem and speech synthesizer really prove attractive to a first-time user? Atari marketing obviously thinks so. I think that people who know they want those features will also know enough to want a disk drive.

Anyway, all of that is crystal-balling and nit-picking on my part. The new lineup of computers is one that any company could be proud of. Atari should be doubly complimented after the fiasco with the 1200.

The New Disk Drives

Before I stop making observations about Atari, though, I would like to carp a bit about one thing: the new Atari disk drives and DOS III (or is it DOS 3?). When I first heard that Atari was going to throw away a potential 50K per disk drive, I thought there was an almost-good excuse. After all, Atari DOS 2.0S could, with absolutely minimum modifications, utilize all the sectors of the one-and-one-third density 1050 drive, so the change, though inefficient when compared to true double-density drives, would allow many current programs to work without modification.

It is not to be. Atari DOS III is just as different from DOS 2.0S as our own Version 4 OS/A+ is. Which means many, many programs (including data base programs, etc.) simply will not work without modification. I do *not* feel this is inherently bad. Let's face it: DOS 2.0S is not a particularly good DOS and it is totally inadequate for larger disk drives. DOS III is actually a very nice DOS for small drives (say up to 128K per drive). It goes downhill rapidly when used on larger drives. This means that if you convert your programs and data files from DOS 2.0S to DOS III this year, you will have to convert to some other DOS again next year, when you move to one of those nice little hard disks I mentioned.

Anyway, when the 1050 finally appears, watch here (I hope) for instructions for using DOS 2.0S (or OS/A+ Version 2) in one-and-one-third density mode, so you won't have to convert all your programs. (You'll still have to convert the diskettes themselves, which won't be easy or fast if you only have one drive, but the same holds true of DOS III – and, to be fair, OS/A+ Version 4 – so you won't have lost anything.)

Self-Relocatable Machine Language, Part III

This month, I will discuss some more techniques which can be used to make your machine language self-relocatable. Last month, we noted which kinds of instructions were implicitly "safe" (register-only instructions, branches, etc.). There was also a list of "Safe Relocatable Techniques." To summarize, the safe techniques mentioned were:

1. Change JMPs to branches.
2. Save register values in the stack, not in fixed memory.
3. From BASIC, pass the address of a string as a location (or series of locations) to load from or store to.
4. Move code from relocatable memory to fixed memory temporarily.

I also promised to discuss two points this month: (1) where the "safe" locations in Atari memory are; and (2) some special techniques usable only with Atari BASIC. Let me fulfill my promise.

Safe Locations

There are none. Next topic.

Oh, all right, I admit that is a bit of an exaggeration, but it is dismayingly close to the truth. When I write machine language routines, I really do prefer that they be usable with as many products as possible. Just as a start – and *not* as a comprehensive list – I would hope that they would work with the following software: Atari BASIC, Atari DOS, OS/A+, BASIC A+, Atari Microsoft BASIC, *Atariwriter*, Atari Assembler Editor Cartridge, MAC/65, AMAC, and a few more.

Okay. Not too long a list. How many zero page locations are not used by any of those? None. How many Page Six locations (\$600 through \$6FF) are not used by any of those? None. How many.... But I think you get the idea. Is all this strictly true?

Actually, there are quite a few bytes which can be used for *your* temporary storage. And I suggest you consult your *Atari Technical User's Notes* or *Mapping the Atari* (from COMPUTE! Publications) to find where they are. (Caution: Watch out for changes in the new XL computers.) But even these locations are suspect. What happens if I write this neat new printer-spooler routine which uses location \$00 (believe it or not, that's free in almost all the above programs), and then you come along and add a driver for graphics mode 27 and you use location \$00?

Perhaps I am being a bit of a purist here. Certainly very little of my own programming is this clean, this free of conflict with other potential programs. And yet it really does require only a little more work to write a program "correctly" (by my definition), so why not do it right? Let's try.

So, we must assume that *no* location outside our own, self-relocatable, properly-loaded-at-LOMEM program is safe at all times. Unpleasant. However, that does *not* say that we can't use some almost-safe locations while our routine has control. In particular, you should be able to use several reserved locations in zero page (for indirect-Y pointers, etc.) by, if necessary, moving values into them from within your relocatable block; using and/or changing the zero page locations in your program; and then moving the values back into your relocatable block.

Sounds complicated? It is. And yet you might be surprised at how seldom you really need to go through all that.

So what zero page locations are safe, even as temporaries? Probably the safest spots, as long as you aren't writing an interrupt handler, are those locations used as temporaries by the DOS File Manager. Locations \$43 through \$49, inclusive, are always reinitialized by FMS every time it gets control. FMS does *not* presume the locations have maintained their contents from one call to the next. (In fact, the locations should properly be called "Device Driver Zero Page Temporaries," since that is what they were intended for.)

And one more comment before I leave you with the impression that absolutely nothing is safe to do on the Atari computers. If you are writing routines specifically designed to be used with Atari BASIC (as I suspect the majority of you are), there are several safe temporaries. First, you can always use the floating point work area, \$D4 through \$EF, whenever BASIC calls either a USR routine or an I/O routine. Also, BASIC does not use locations \$CB through \$CF (only four bytes!) at all. Again, let me give you the caution about adding your routine to a system which already has a custom routine. Be sure there is no conflict.

A Built-In Relocatable Pointer

It's true. There really is such a thing. There are some *ifs* though: if you are using Atari BASIC or OSS BASIC A+ or OSS BASIC XL; if you have placed your relocatable program in a string and are calling a machine language routine via USR(ADR(STRING\$)) or USR("...machine-language-string..."); if you don't mind a small trick.

First, the trick. It's really quite simple. Whenever BASIC calls a USR routine, it calls the routine by placing the routine's calculated address in location \$D4-\$D5 (which just happens to be the first two bytes of floating point register zero). It then JSRs to a routine which simply does a "JMP (\$D4)", a jump indirect to the USR routine.

But why can't we take advantage of that pointer? It already points to our relocatable program, so why can't it point to our relocatable data? Perhaps a demonstration is in order.

```
FR0 = $D4
USRROUTINE
    CLC
    BCC START ; branches are ok
;
SAVEBYTE .BYTE 0 ; some data
;
; begin actual code
;
START
    LDY #SAVEBYTE-USRROUTINE ; index
    PLA ; count of parameters
    CMP #1 ; how many?
    BNE NOPARAMS ; none, we presume
; the user is passing a byte to us
    PLA ; high byte...ignored
    PLA ; low byte...stored
    STA (FR0),Y ; thusly
; we join here, whether a byte is passed or not
NOPARAMS
    LDA (FR0),Y ; get the byte
    STA FR0 ; to be returned
    LDA #0
    STA FR0+1 ; high byte zero
    RTS
```

This program is a *very* dumb one, for demonstration purposes only. If you call it from BASIC via, for example, "PRINT USR(routine)", your program will print the byte value saved in location SAVEBYTE (zero, initially). On the other hand, if you use "JUNK=USR(routine, 97)", the routine will store the second parameter (97) in location

SAVEBYTE. Presumably, you could then later recover the 97.

The point to be made, however, is that this program is completely self-relocatable and yet is able to load and store data from within its own relocatable block! The secret is the "LDY #SAVE-BYTE-USRRoutine" line directly after the label START. Since location FR0 contains the address of USRRoutine, loading the Y-register with the proper offset (SAVEBYTE-USRRoutine, which happens to be 3 in our example) will allow us to do indirect loads and stores to any location within 255 bytes following USRRoutine.

Can I put that more clearly? Since, when we do either the "LDA (FR0),Y" or the "STA (FR0),Y", the Y register contains the value 3 and location FR0 points to the location USRRoutine, the LDA and STA instructions will reference the third byte after USRRoutine. Which just happens to be SAVEBYTE.

And just a reminder if you don't know or remember what the PLA instructions in this program are for. Whenever BASIC calls a USR routine, it pushes all the parameters it is given onto the CPU stack (after first converting them to 16-bit integers, of course). Then, the last thing it does before the call is to push a count of the number of parameters (presumed to be 1 or 0 in our example) onto the same stack. Thus, the first PLA lets us

discover how many parameters were passed. The other two PLAs are necessary if a parameter is passed; otherwise the RTS instruction will return to an unknown location and will likely crash the system. (Note that in our simple-minded example you can probably crash BASIC by calling the routine with two parameters, since no check is made for more than one parameter.)

Next month we're going to take this technique a couple of steps further. We will discover how to have more than 255 bytes of relocatable storage (which may or may not be useful to you) and how to generate similar self-pointers when the routine in question has not been called from BASIC. **C**

ATARI

GRAPHICS HARDCOPY

NOW FOR NEC & OKIDATA

Dumps anything on the screen of an ATARI 400/800 to a printer. All graphics & text modes. Players/missiles/scaling/grey scale/GTIA/more! Works with EPSON, NEC, Okidata, Centronics 739, IDS and Trendcom. Specify 800 or 400 and printer when ordering.

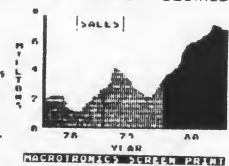
INCLUDES CABLE & SOFTWARE
850 MODULE NOT REQUIRED



(209) 667-2888

MACROTRONICS, Inc. C.O.D.

1125 N. Golden State Blvd.
Turlock, California 95380



MACROTRONICS, INC. PRINT

*ATARI is a registered trademark of ATARI Computer Inc.

ATARI® 400® AND 800® OWNERS

Question #6:

How can you have 64K RAM and complete compatibility with all Atari products?

- A. Weld 3 computers together
- B. Drugs
- C. The Mosaic 64K RAM Select
- D. Exercise
- E. All of the above

Answer: THE MOSAIC 64K RAM SELECT™. It's the most advanced memory system of its kind available. For more information and your nearest MOSAIC Dealer, call 1-800-547-2807.

COMPILE ATARI BASIC AND FLY!



With ABC™, Monarch's new BASIC compiler for ATARI 400 and 800, you develop and debug programs using your ATARI BASIC cartridge, then use ABC to transform them into compact code that runs up to 12 times faster, without the cartridge (and protects your source code, too). 40K and disk required. For your ABC diskette and manual, send check or money order for \$69.95 (or \$9.95 for manual alone).

Monarch Data Systems

P.O. Box 207, Cochituate
MA 01778, (617) 877-3457.



Mastercard/Visa by phone. Dealer inquiries invited. Mass. residents add 5% sales tax. ATARI, ATARI 400, and ATARI 800 are trademarks of ATARI, Inc.

INSIGHT: Atari

Bill Wilkinson

Last month, I said that this month's column would include the final part of the series on writing self-relocatable code. Unfortunately, that project has turned out to be bigger than I thought it would be, so I am going to put it off a month and devote an entire column to it. However, as compensation, I will finally discuss the "new" Atari graphics modes I hinted at a couple of months back. Before I get to the juicy stuff, though, I'd like to continue a little of the ranting and raving that I started last month.

How To Shed 50 Kilobytes Without Even Trying

I heard (from two different sources) the official Atari "line" regarding the new 1050 disk drives. It seems that Atari chose to utilize only 128 bytes per sector and only 127K bytes of file space per drive in order to achieve "increased reliability." Honest. Do you believe it?

Actually, that's pure computer puckey (to paraphrase Colonel Sherman Potter). And it's ridiculous for several reasons.

First off, Atari is implying that double-density drives are unreliable. If that's true, then IBM, Radio Shack, Commodore, and a lot of other computer companies are in real trouble. Actually, Atari and Apple are the only major computer companies still relying on single-density technology as their primary *modus operandi*. And, despite Atari's claims, even Atari's 1050 is actually using true double density.

It turns out – based on what we have gleaned from the specs of DOS III at this time – that Atari formats the 1050 drive with 40 tracks of 32 sectors each, with 128 bytes per sector. That's a total of 160K bytes. Most double-density manufacturers achieve either identically the same total (40 tracks times 16 sectors times 256 bytes) or slightly more (40 tracks times 18 sectors times 256 bytes equals 180K bytes – the format used by most Atari-compatible drives such as Percom, Astral, Micro-Mainframe, etc.). So why does Atari claim only 127K bytes?

Real simple: DOS III only supports 127K bytes. Shall I say that I don't know why Atari chose this limitation? With a relatively minor modification, and by using only another 64 bytes of memory per drive, DOS III could have supported a full 180K drive.

Now, as it turns out, I do happen to know the real reason Atari chose 128 bytes per sector.

And I know this from the most reliable of sources, one of the DOS III's designers.

It seems that so many of Atari's own products violate Atari's own "rules" (especially those about respecting the LOMEM pointers), and so many other products also reach outside DOS to do direct sector disk I/O that Atari's planners were fearful of the impact of changing either LOMEM or the sector size. Hence the scheme of DOS III.

A secondary impact of the LOMEM problem was that it caused more and more of DOS III to be moved to the diskette from memory, to be called in as overlays when the user requested a function not in memory. Even the keyboard menu processor eventually got moved to disk. The result of all this? While DOS III may be the easiest-to-use DOS yet, it still suffers from the time-consuming swaps to a MEM.SAV file when you want to achieve something as simple as getting a disk directory.

(Of course, there is a very, very elegant way to completely avoid the LOMEM problem on the new Atari XL computers. Why not move the DOS into the as-yet-unused extra memory? Why waste 14K bytes of RAM? I probably shouldn't drop this idea in Atari's laps [I should sell it to them], but it will take them at least six months to even discuss it, so I figure it's OK.)

As I said last month, DOS III contains a nice little file manager. It's a crying shame that it wasn't released three or four years ago, since it seems ideally suited to an 810 size drive. But it doesn't look to me like a system for the long haul, when larger and larger drives become available for the new Atari computers.

And lest too many of my critics cry "foul" for my promoting OSS's version 4 DOS (which will allow up to 32 megabytes per disk drive), let me hasten to say that I am *not* suggesting that version 4 and the 1050 are necessarily the answer. What I *am* saying is that Atari could have achieved virtually the same results by sticking with DOS 2.0 and extending it to handle up to 120K bytes of file space (with 128 byte sectors – it will handle 240K bytes with 256 byte sectors).

Well, enough. I promise no more on this subject until I give you the patches to DOS 2.0s to give you 120K bytes on a 1050. In the meantime, ask yourself this question: if DOS III is limited to 127K bytes of file space, how will Atari handle the double-sided, double-density drive in the 1450XLD, which will have a capacity of at least 320K bytes? Atari, will you answer?

One more comment. I just want to say that, aside from the 1050, I am impressed with all of Atari's new hardware products. And I even like some of their new software. I think Atari is back on its feet and running hard.

Four Equals Seven

Many of the games currently on the Atari market use custom-designed character sets for background displays. The classic example of this is, of course, *Eastern Front* by Chris Crawford. That beautiful scrolling map he displays is actually composed of "characters." This works because a couple of the ANTIC graphic modes allow the programmer to treat each pair of bits within a character cell as one of four colors.

In fact, by controlling the high order bit of the character to be displayed, the programmer may choose two different sets of four colors. Which would be really nice except for the fact that only one of the colors can change between the two sets, thus there is a total of five displayable colors.

If you don't remember and/or understand all that, don't worry. There's a better way. A way which will get you seven colors! The method only works on machines with a GTIA installed, but I hope that all *COMPUTE!* readers have added a GTIA by now. (If you have purchased a machine in the last year and a half or so, you got a GTIA with your machine. If you have an old machine with a CTIA, the upgrade cost is nominal.)

The credit for finding and documenting this until now hidden feature of the Atari must go to Steve Lawrow, the author of our MAC/65 assembler. He did a nice job of investigating all the ramifications and provided me with the table which I've reproduced here. Before I go into the details of the table, though, let me briefly describe how one accesses two new Atari Graphics modes.

Getting At The New Modes

First, the new modes are variations on BASIC GRAPHICS 1 and GRAPHICS 2 (and, by extension, GR. 17 and GR. 18). And the method of producing the variations is so simple that it's almost funny that no one stumbled on it before. Simply turn on the GTIA's special color mapping mode. And what, pray tell, is that? In this case, it is the upper bit of GPRIOR, the priority select register.

GPRIOR is a hardware register that has its OS shadow location at \$026F (decimal address 623). That means (for those of you not familiar with shadow locations) that by changing the RAM location \$026F you cause the OS to change the appropriate hardware register for you. (And see *COMPUTE!*'s book *Mapping the Atari* if you need to know more.)

Briefly, then, you need simply to turn on the upper bit of GPRIOR in order to activate these new modes. There are, however, some caveats to be observed. Perhaps the easiest way to observe the toughest potential problem is to turn on your Atari, go into BASIC, and do a POKE 623,128.

What do you see? Garbage on the screen, if you have a GTIA. Unfortunately, activating the GTIA destroys the normal character display mode(s). More on this later.

Now, on to the table. When you tell BASIC to PRINT #6 in Graphics modes 1 and 2, it prints larger than normal characters to the upper portion of the screen. In particular, though, the characters are available in several different colors. Try this little program to see what I mean: GRAPHICS 2 : PRINT #6; "AaAa" (where the underlined characters are typed in inverse video).

And why do you get four different colors? Because the upper three bits of each of the characters are different. In particular, the upper three bits for the four characters shown are 010, 011, 110, and 111, respectively. Because you are in Graphics mode 2, all four characters came out as uppercase letters.

Now, the bytes which are put in screen memory are actually translations of the bytes which you PRINTed. In particular, when the bytes shown are translated to screen codes, they end up with upper bits of 00, 01, 10, and 11, respectively. The upper two bits of the bytes placed in screen memory determine the color to display; the bits in the character set determine which bits will be "turned on" on the screen.

The concept used in our "new" graphics mode is similar. In particular, the upper two bits of the bytes placed in screen memory determine the color MAP to use. The actual bits in the character set determine which color will be selected from the appropriate map. In other words, we have added yet another level of color indirection to the Atari!

In GRAPHICS 10, memory is organized in groups of four bits. The value of the four-bit nybbles determines which color register is displayed. Thus, since there are nine color registers (five for the primary graphics and four for player/missile graphics), there are a maximum of nine independently displayable colors. (Yes, I know that you can get 16 colors in GRAPHICS 9 and 11; but in those cases the colors are not truly independent.)

In GRAPHICS 1+ and 2+ (well, I had to call them something, didn't I?), pairs of bits (instead of four-bit nybbles) determine the color register to use. Remember, though, that the pair of bits can only select a color from the particular MAP which has been selected by the two upper bits of the character on the screen.

And, finally, this implies that the other six

PLAY THE ARCADE GAMES THAT TALK BACK!

What do Program Design games include for free that other companies charge you hundreds of dollars for?

The human voice.

Only Program Design software comes with a voice based cassette synchronized with your Atari computer. So now, you not only use your eyes and intellect to play one of our arcade games. You use your ears too. Visit your nearest software dealer and take a test listen.

DON'T PLAY 'CLIPPER' IF YOU CAN'T SWIM!

You're the captain of the clippership Flying Cloud. And there's never been a tougher sea challenge than your voyage from New York to San Francisco via Cape Horn.

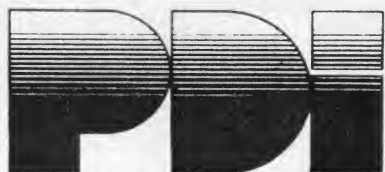


As you cast off, the actual sound of old sea chanteys fills the air. But there's no time for singing, because you have to navigate through storms and icebergs. As if that wasn't enough, there's the constant danger of being thrown over board by a mutiny, so you better know how to swim. Remember, you're in charge. So pick your cargo, crew and course very carefully.

DEFEND 'MOONBASE 10' AND THE PRESIDENT WILL THANK YOU PERSONALLY

The voice of mission control asks you to defend Moonbase 10 from a horde of alien invaders. But first, you and your robot must navigate through mine fields. Moonbase 10 combines three adventure games in one. And when your mission is completed the president will thank you personally (so mind your manners). Moonbase 10 is the winner of the most innovative game award from Electronic Games magazine.

Clipper comes in 32K disk, cassette & joystick or 24K cassette & joystick. **Moonbase 10** comes in 24K disk, cassette and joystick or 16K cassette & joystick.



Program Design, Inc. 95 East Putnam Avenue,
Greenwich, CT 06830

*Atari is a trademark of Atari, Inc.

bits of the character on the screen select the particular character from the character set memory, just as normal GRAPHICS 1 and 2 do.

Does that sound complicated? It should, because it is. Anyway, now is the time to look at the table. It shows the MAPs that are available.

Color Selection Under GRAPHICS 1+ And 2+

Bit Pairs Of Color Selectors In Character Memory

Upper Bits of Character on Screen (Map Selector)	00	01	10	11
00	704	704	704	704
01	704	705	712	709
10	704	706	712	712
11	704	707	712	711

And, you presumably ask, what are the numbers shown in the table? Simply the location of the color register which will be displayed when you use the given bits within the given map. For example, 704 is PMCOL0 (player color 0) and 712 is PFCOL5 (playfield color 5). However, the easiest way to change the color registers, in this instance, might be to refer to them via the locations shown in the table.

So, writing POKE 704,0 will make the background color black. Writing POKE 712,152 will give you a nice blue for bit pattern 10 in MAPs 01, 10, and 11. A little observation of the table will show you that MAP 00 is essentially useless: it always gives you the background color, regardless of the bit patterns in the character memory.

On the other hand, bit pattern 00 always gives you background color, regardless of the MAP used, so it may prove useful in many circumstances. For the rest, note that MAP 10 gives you only three colors, but it is the only MAP which gives color 2 (706). Sigh. The system is not totally flexible, but it is handy.

First thing next month we'll put this all together with a little BASIC program that demonstrates the capabilities of the new modes. ©

Use the handy
reader service cards
in the back of the
magazine for
information on
products advertised in
COMPUTE!

COMPUTE's

The Atari BASIC Sourcebook

Authors: Bill Wilkinson,
Kathleen O'Brien, and
Paul Laughton

Price: \$12.95

On Sale: Now

If you program in BASIC, you know about commands like PRINT, GOSUB, IF-THEN, and others.

But did you know that each of these commands is actually a mini-program in itself? Atari BASIC is a collection of machine language routines that tell the computer what to do, how to do it, and what to do next.

Atari BASIC is a powerful and versatile language. Now available from **COMPUTE! Books**, *The Atari BASIC Sourcebook* offers Atari programmers a chance to look inside the language — directly to the source code that is Atari BASIC.

Authors Bill Wilkinson, Kathleen O'Brien, and Paul Laughton, the people who wrote Atari BASIC, take you on a tour through the language. They explain how it works and how you can make it work for you.

The Atari BASIC Sourcebook answers these questions (and more):

- When you RUN a BASIC program, what is really going on inside the computer?
- How does the computer know how to handle a FOR-NEXT loop? How does it RETURN from a subroutine?
- Where do ERROR messages come from? How does the computer know what's wrong?
- How does your Atari decide which mathematical operation to perform first?
- Why do some tasks take so long, while others happen almost instantly?
- Why does the computer sometimes lock up when you delete lines from a program?
- How does the computer interpret words and symbols like GOTO, INT, CHR\$, *, and =?
- How can a machine language programmer take advantage of the sophisticated routines in Atari BASIC?

Intermediate to advanced Atari programmers will find a wealth of useful and interesting information in *The Atari BASIC Sourcebook*.

Much more than a simple source code listing, this book explains how BASIC works and why. All major routines are examined and explored. The authors go into detail about the internal design, the stack, input/output statements, and much more. When you finish reading this book, you will have an in-depth understanding of how to put Atari BASIC to work for you in ways you never thought possible.

The Atari BASIC Sourcebook is available at many computer stores and bookstores, and can also be ordered directly from **COMPUTE! Books**.

Available at computer dealers and bookstores nationwide. To order directly call TOLL FREE 800-334-0868. In North Carolina call 919-275-9809. Or send check or money order to **COMPUTE! Books**, P.O. Box 5406, Greensboro, NC 27403. Add \$2 shipping and handling. Outside the U.S. add \$5 for air mail, \$2 for surface mail. All orders prepaid, U.S. funds only.

Publisher's Foreword	v
Acknowledgments	vii
Preface	ix

Part One: Inside Atari BASIC

1 Atari BASIC: A High-level Language Translator	1
2 Internal Design Overview	7
3 Memory Usage	13
4 Program Editor	25
5 The Pre-compiler	33
6 Execution Overview	49
7 Execute Expression	55
8 Execution Boundary Conditions	71
9 Program Flow Control Statements	75
10 Tokenized Program Save and Load	81
11 The LIST and ENTER Statements	85
12 Atari Hardware Control Statements	91
13 External Data I/O Statements	95
14 Internal I/O Statements	103
15 Miscellaneous Statements	105
16 Initialization	109

Part Two: Directly Accessing Atari BASIC

Introduction to Part Two	113
1 Hexadecimal Numbers	115
2 PEEKing and POKEing	119
3 Listing Variables in Use	123
4 Variable Values	125
5 Examining the Statement Table	129
6 Viewing the Runtime Stack	133
7 Fixed Tokens	135
8 What Takes Precedence?	137
9 Using What We Know	139

Part Three: Atari BASIC Source Code

Source Code Listing	143
---------------------------	-----

Appendices

A Macros in Source Code	273
B The Bugs in Atari BASIC	275
C Labels and Hexadecimal Addresses	281
Index	285

INSIGHT: Atari

Bill Wilkinson

Bill concludes last month's column with a program demonstrating the capabilities of the new graphics modes.

If you were a little disconcerted by our discussion last month, here is a little BASIC program which demonstrates the capabilities of the new modes in a crude, but visible, fashion. As usual, I will explain the program line by line.

120. Selects a normal GRAPHICS 2. This is our starting point.

130. Prints a reference line on the screen. This is simply so you can tell where the columns of characters are later, when they get MAPped.

150-180. Print what are now normal characters. Note that the underline denotes inverse video characters (via the Atari key). Did you notice that each set of four characters here will produce the MAP patterns 00, 01, 10, and 11 (in that order) on each line of the displayed area? Remember that the other six bits, then, will select a character from character memory.

190, 290, 320, and 340. Just messages, to tell you what we are doing.

200-220. We are moving the normal Atari 800 character set from its normal location (\$E000) to RAM at address \$6000. Note: This requires a 32K machine.

230-250. Here we read the DATA statements from lines 380 to 420 and change the character set for the characters A, B, C, and D.

260-280. A quick and dirty way to arbitrarily select some colors for the various color registers.

300 and 330. Just some delay loops, so you can actually see it happening.

310. Changes the CHBASE (CHaracter BASE pointer) to point to location \$6000, where the new character set pattern is.

350. The magic instruction. Look at your screen. How many different colors do you see?

Do you see the relation between the display and the table? Did you notice that the first character in each line "disappeared"? That's because these characters are using MAP 00, the "all background" map.

I think the only thing left is to explain the bit patterns of the modified characters which are read in by lines 230 to 250.

Character A is changed to a solid block of all

"11" bits (thus the pattern is eight \$FF bytes).

Character B is changed to a solid block of all "10" bits (eight bytes of \$AA). Character C is a solid block of "01" bits (eight bytes of \$55).

Finally, character D has a purposely varied pattern. The bit patterns in the byte are as follows:

228	\$E4	11	10	01	00
57	\$39	00	11	10	01
78	\$4E	01	00	11	10
147	\$93	10	01	00	11

and then the same bytes in reverse order.

The result of the shifted bit pattern shown is, quite naturally, the "arrows" which you see in the program's display.

Finally, we are finished explaining these new modes. What good are they? Just imagine what Chris Crawford could do with a map which displays seven different colors, instead of only four. But surely there are other uses. How about inventing some and sharing them with us?

```

100 REM DEMO OF THE "NEW" GRAPHICS MODE!
110 REM
120 GRAPHICS 2
130 PRINT #6;"wxyz"
140 PRINT #6;" "
150 PRINT #6;"AaAa"
160 PRINT #6;"BbBb"
170 PRINT #6;"CcCc"
180 PRINT #6;"DdDd"
190 PRINT "THIS IS IN NORMAL GRAPHICS 2"
200 FOR A=24576 TO 25599
210 POKE A,PEEK(A+32768)
220 NEXT A
230 FOR A=24840 TO 28671
240 READ D:IF D<0 THEN 260
250 POKE A,D:NEXT A
260 FOR A=0 TO 8
270 POKE 704+A,18*A+18
280 NEXT A
290 PRINT "THIS IS WITH COLORS CHANGED"
300 FOR I=1 TO 1000:NEXT I
310 POKE 756,96
320 PRINT "THIS IS THE MODIFIED CHARACTER SET"
330 FOR I=1 TO 1000:NEXT I
340 PRINT "FINALLY, THE NEW AND SPECIAL MODE!"
350 POKE 623,128
360 REM == JUST A LOOP TO KEEP DISPLAYING ==
370 GOTO 360
380 DATA 255,255,255,255,255,255,255,255
390 DATA 170,170,170,170,170,170,170,170
400 DATA 85,85,85,85,85,85,85,85
410 DATA 228,57,78,147,147,78,57,228
420 DATA -1
    
```

C

INSIGHT: Atari

Bill Wilkinson

This month I will discuss extended memory management on the Atari computers. Before I start, though, I would like just to chat for a bit. (If you are waiting for the last part of the series on self-relocatable code, be patient. It's just bigger than I expected it to be, so I've got to massage it a bit more.)

Some Small Talk About Computers

Today I read an interview with Alan Kay in *Technology Illustrated*. As many of you probably know, Alan Kay was perhaps the most instrumental person in the development of the Smalltalk language. (Or is it an operating system? Or is it more properly called simply an "environment"?)

The work he did on Smalltalk while at Xerox caused him to believe that computers were destined to become a household tool, as common as, say, the television set. (Which may seem a mundane belief today, but Kay was saying such things five to ten years ago.) Well, Atari apparently liked Kay's philosophy, vision, and capabilities, and hired him awhile back.

The article I read interested me in two ways. First, it labeled Kay "Atari's Chief of Games." Well, I had been led to believe that he had been brought to Atari to head research and development, presumably to lead Atari into the generation beyond Smalltalk (a logical presumption, since he'd stated that he felt Smalltalk had served its purpose, was obsolete, etc.).

Anyway, with my orientation toward languages and systems, I saw "Chief of Games" as a step downward. Yet the interview made it clear that Kay felt he was in perhaps one of the most challenging positions possible. Hmmmm. What has changed? Are games truly the most useful purpose of a computer right now? The marketplace certainly seems to think so. It is food for thought.

The second thing in the article which really got my CPU stirred up was Kay's view of the computer. I had always been under the impression that he believed his real goal in life was to enable

everyone not only to use the computer, but to actually command and manipulate it. (I hesitate to say "program it," but then Smalltalk is a language.) In the interview, though, Kay stated he was beginning to fear that perhaps the computer was not so much a household tool as it was a fine instrument, like a violin. He strengthened the analogy by noting that very few people can play the violin, just as very few people can properly use a computer.

Well, I for one believe that not only is the analogy inappropriate, but its projection of gloom and pessimism about the future of computers is not justified. Granted, the analogy may hold today. After all, only about 1 percent of the United States population can claim to be able to program at all (or play "Twinkle, Twinkle, Little Star" on the violin). Probably less than .1 percent produce acceptable application programs (or play in a community orchestra or equivalent). Dare we guess that .01 percent are commercial programmers (or make their living playing the violin)? Can it be that only .001 percent can actually write systems and languages (or are the guest soloists of the concert world)?

Actually, these proportions are just order-of-magnitude guesses, but they do seem to support Mr. Kay's analogy. But I say that his analogy has validity mainly because the computer is still such a relatively "rare" instrument. Personally, I prefer a different analogy.

When computers are as much a part of everyday life in this country as automobiles are now (and I firmly believe that they will be), then I think they will be treated much as automobiles are.

Let me sidetrack a little. Here in California, the State has decreed that all high school students shall take a course in "computer literacy." So what happens? Every high school is scrambling to buy one or two computers and begin teaching every kid how to program in BASIC. Great, right? *Nonsense!*

Two Different Classes

First of all, I can't conceive of learning how to use

or program a computer at all if the student/computer ratio is above 3 to 1. More importantly, I think it is senseless to equate "computer literacy" with "learning to program in BASIC." After all, "automobile literacy" consists of learning traffic laws, safe driving techniques, and actually starting to drive a car (it's usually called "Driver Training").

"Automobile expertise," on the other hand, consists of learning what tools do what, the theory and practice of internal combustion engines, and how to maintain and repair an automobile (and this is usually called "Auto Shop"). Does every student take driver training? Yes, or nearly so. Does every student take auto shop? No. Not by a long shot.

So, I believe, it should be with computer literacy. Don't teach everyone how to program. (What would we do with a nation of programmers? The same thing we would do with a nation of auto mechanics?) Instead, teach everyone how to use a computer to do word processing, to balance their budget, to access data bases, and the list could be quite long.

And, yes, keep the computer programming classes. But keep them on the same basis that auto shop classes are offered — as electives, for those interested in learning more than how to "drive" their computers or cars.

Why this confusion of computer literacy and computer expertise among schools and teachers? Partly because the computer industry has promoted the view. (Perhaps fearing that current applications programs are inadequate to a classroom situation?) Partly because of a dismal lack of education and information on the part of the educators. (Pity the poor math or history teacher who is nearing retirement. Suddenly he/she is forced to learn enough about these nasty machines to be able to teach some kids how to use it. Do you wonder that the path of least resistance is most often chosen?) Mostly, I suppose, because BASIC comes built into each machine, while good text processors, spreadsheet programs, etc., cost extra, money which most schools don't have.

So how does this tirade relate to either Alan Kay or you, my patient reader? Well, first of all, I think the analogy of car and computer is a better one than violin and computer. And, perhaps, if computer companies started trying to design mass consumable "cars" instead of trying to ply the public with precision instruments, it is a future that will come true. To be fair, I think that companies such as Atari and Commodore and Apple and others are starting to do so already. But my cynicism leads me to believe that they are driven by the current market, not by the future one.

You're Ahead Of Your Time

Perhaps more importantly, though, I am trying to convey the message that those of you who read

this column (and this magazine) are, in some sense, ahead of your time. You are, indeed, the violinists that Alan Kay perceives. Some of you are just learning to play your first notes. Others of you are already tackling the great concertos. But, when the computer revolution really arrives, you will all have the advantage of having already taken at least your first "auto shop" course. So, if you enjoy your computer (and particularly if you enjoy programming), don't give it up easily. And certainly don't give it up now. Someday, others will appreciate your art, however humble or glorious it may be.

Did that sound like a sermon? If so, I apologize. But it's my view of both the present and the future of computers and programming. One last sidelight before we move on: On hearing me espouse the views above, someone once asked me what my position in the hierarchy was, as a person who helped design (as opposed to program) operating systems and first languages for new machines. Actually, that's an easy question: I'm simply a composer. And so, I think, are such people as Alan Kay.

You Can Bank On It

All of the new Atari XL computers (including the 1200XL) will contain 64K bytes of RAM (the 600XL requires an external RAM pack to do so). And all contain 16K bytes of Operating System ROM space. And, further, all (except the 1200XL) include good old Atari 8K BASIC. Let's see here — 64K plus 16K plus 8K — that's over 90,000 bytes of space.

Wait a minute, though. If I plug in a 16K cartridge (such as AtariWriter or ACTION! or BASIC XL), then I could have 104K bytes of RAM and ROM. Wow. That's really nifty, right? Well...

Have you read this column often enough to know that "Well..." means "not really" or "there's more to come"? No? Well...

Not really. To begin with, all Atari computers are built around the same CPU (Central Processing Unit), the 6502. (Which, incidentally, is the same chip used in most Commodore computers and all Apple machines except the Lisa.) However, there is a fundamental restriction involved when using a 6502: There is simply no way to access more than 64K bytes (65,536 bytes) at one time. How, then, can the Atari use 104K bytes? Is someone fibbing to us?

The key here is the phrase "at one time." A juggler may be able to juggle only four things at a time. Does that mean he always juggles the same four objects? Should we presume that the 6502 must always work with the same 64K bytes? Of course not.

In point of fact, the new XL machines allow the 6502 a number of choices about which bytes it will "juggle." How the 6502 makes its choice is

the subject of this section.

Actually, there is no magic formula or scheme which enables the various choices. In fact, various choices are made by differing means. Generally, the choice is "consciously" made by the program currently in control of the machine. And it makes the choice simply by (usually) storing something in a particular memory location. Confused? Let's digress a little.

Some CPUs (including microcomputers and minis and maxis) treat input/output as a separate domain from general memory. For example, the 8080/Z-80 group of processors allow up to 256 separate input and output ports, which are completely separated from the general RAM/ROM memory (they even have special instructions specifically for reading/writing these I/O ports). On the other hand, many machines (such as the 6800, 68000, and 6502 families, as well as such giants as the PDP-11 series) simply treat input/output ports as part of the general machine memory.

Efficient And Easily Learned

The advantages and disadvantages of each scheme are a subject of hot debate, but I will only present a single aspect of each here: Keeping the I/O ports out of general memory allows a true 64K bytes of RAM when using an 8- or 16-bit microprocessor. Allowing I/O to be treated as part of memory means that any instruction which can access RAM or ROM can also access a port, often resulting in efficient and easy-to-learn coding.

Anyway, note that the 6502 does, indeed, use what is called "memory mapped I/O," and Atari computers do, as a consequence, reserve 2K bytes of memory (addressed from \$D000 to \$D7FF) which is specifically designed for I/O port addresses. (If losing 2K of your space seems excessive, pity the Apple owner who loses 4K.)

In the case of the XL machines, then, one simply changes the value in an I/O port — which appears to one's program as a memory address — and presto, a different choice of "jugglable" memory is made. But what I/O port to use? Did you notice the fact that Atari 400 and 800 computers have four joystick ports while the XL machines have only two? Guess which ports are now used for memory juggling. Did you need more than one guess?

For the more hardware-oriented of you out there, I will note that all four Atari joystick ports are actually nibble-sized pieces of a 6820 (or 6520) PIA (Peripheral Interface Adapter). The PIA is a very flexible chip; it allows each of its 16 I/O pins to be separately configured to be either an Input line or an Output line. In the case of the 400 and 800, all 16 lines are configured as Input, since they are all used to read the four directional switches of an Atari joystick. In the case of the XL

machines, some of them have been changed to Output lines, thus enabling them to act as electronic switches.

On the 1200XL, for example, two of them are used to control the L1 and L2 status LEDs. And (you saw this coming, I presume) two of them choose certain configurations of the computer's memory. (On the other XL machines, still another line is used to control still another possible configuration.)

Since we are discussing memory configuration choices, I might as well confuse the issue a bit more by also mentioning how we at OSS implemented our new SuperCartridges. It is probably no accident that Atari provides the cartridge slot on all machines with a line labeled "CARCTL", an abbreviation for CARtridge Control. Actually, this line is active whenever any memory location from \$D500 to \$D5FF is accessed. Since no Atari cartridges take advantage of this line, we thought it was time that we did so.

One At A Time

About now, it is past time for a diagram. The figure shows all the possible choices of memory configuration by placing them in memory address order. Note, though, that the 64K addressing restriction of the 6502 applies. Hence, when two or more choices are given for a particular address range in memory, remember that only one such choice may be active at any given time. For each address range where a choice is available, there are two or more *banks* of memory. And choosing one bank over another is called *bank switching* or *bank selection*.

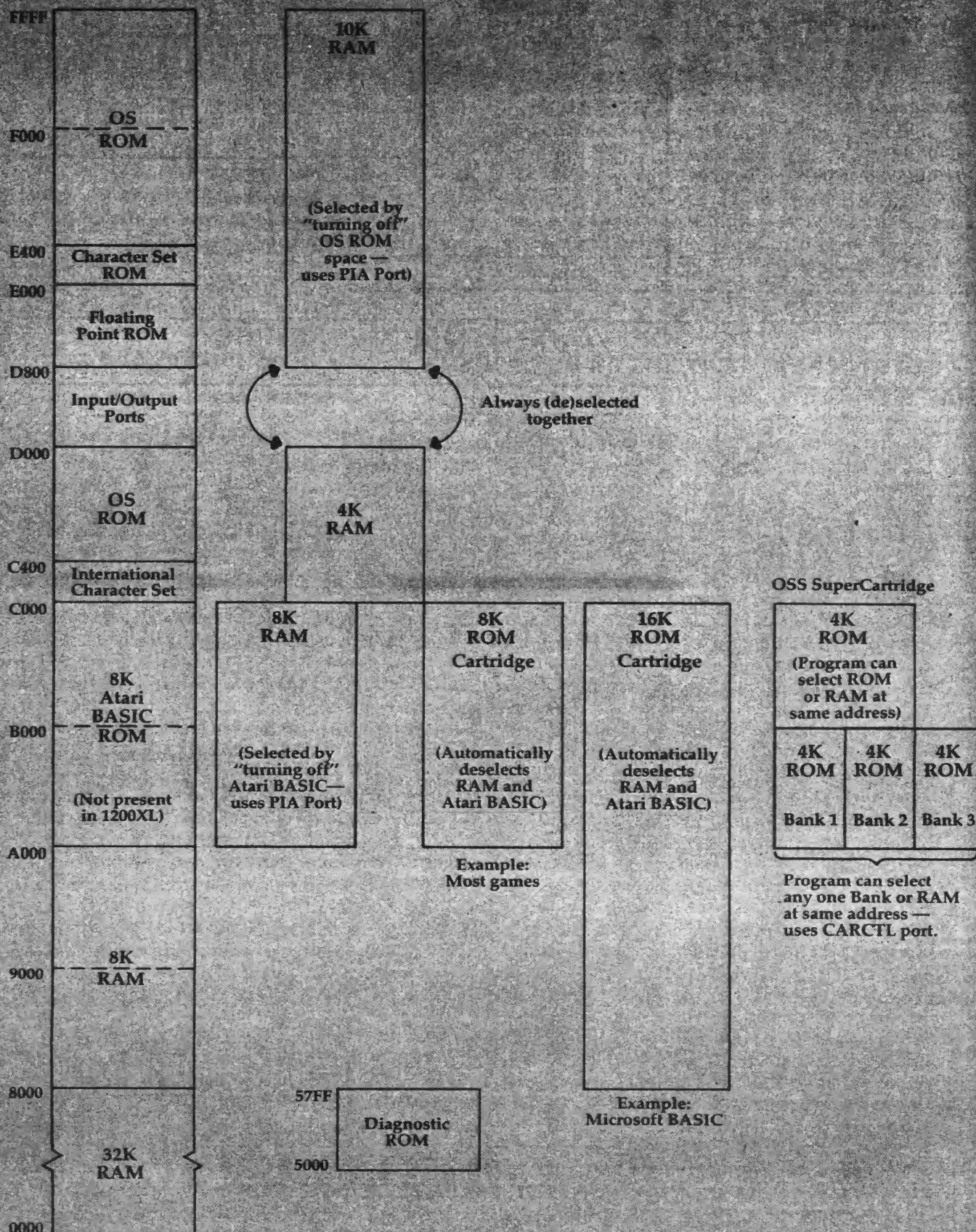
For example, I might choose to use BANK1 of the SuperCartridge while at the same time choosing the RAM BANK of system memory. The important thing to note here is that each set of banks (that is, parallel memory segments), as shown in the figure, is independently bank selectable.

Also, some bank choices are not available at the software level. For example, when you plug in a Microsoft BASIC cartridge, you have 16K bytes of ROM from \$8000 to \$BFFF. You have no RAM in that address range. You have no choice in the matter. This is, then, hardware bank selection.

The advantage of hardware bank selection is that it is essentially foolproof. If the hardware removes a bank of RAM from your program's "vision," your program can't get into trouble trying to use that bank.

But the advantage of software-selectable banks is, quite simply, that they allow you to expand the capabilities of your machine. If you look at the figure, you can see that a SuperCartridge allows you 16K bytes of programming power while occupying only two 4K byte banks at any given time.

Memory Map Of Atari XL Computers (Showing Parallel Memory Banks At Same Addresses)



And the purpose of this discussion? To show that the XL machines really do have a lot of latent power. How do we make it un-latent? Well....

As I write this article, the number of commercially available programs which allow you to take advantage of the extra 14K bytes of RAM on an XL machine is countable on the fingers of my left foot. Zero. By the time you read this, there will likely be products heading your way that will justify the purchase of an XL machine (or a 64K memory board, such as the one from Mosaic Electronics, for your 800).

Since I am obviously most familiar with DOS XL, let me explain a little of how it works.

When DOS XL boots into an XL computer, it first establishes a set of jump vectors for the various interrupt routines. Why? Because any IRQ, NMI, or SYSTEM RESET will attempt to jump through the vectors which must (by 6502 CPU law) be located at addresses \$FFFA through \$FFFF. If we deselect the OS ROM bank in order to enable the RAM bank at the same addresses, the contents of these critical addresses are unpredictable. We *must* supply some valid routine addresses or the system will crash.

DOS XL puts most of the DOS code in the RAM bank which is "under" the OS ROMs. It also leaves a piece of itself at the conventional DOS load address of \$700 (an area of memory which is not bank selectable). Then, if there is a BASIC cartridge in the machine, it selects the OS ROM bank and jumps to BASIC.

So long as BASIC makes no calls on DOS, all is calm and expected. However, watch what happens when (for example) we try to open a file from BASIC.

1. BASIC sets up an IOCB with a pointer to the filename. Since the filename was specified by the user, the pointer will contain an address somewhere between about \$A00 and \$9C00. BASIC makes a call to \$E456, the CIO entry point.

2. CIO determines that the device requested is actually the disk file manager and uses the "D:" device table to determine the address of the disk's open file routine. It passes control to that routine.

3. Note that the "D:" device table and at least the first part of the file open routine must be in nonselectable RAM (that is, at or near \$700). The file open routine is a big one, so it selects the DOS XL RAM (disabling the OS ROM) and jumps to the main part of the code.

4. The main code is able to examine the filename since it is in nonselectable memory, so the file open is performed if possible. The main code exits back to the tail end of the OPEN code, near \$700.

5. This tail end then simply reselects the ROM bank and returns to where it was called (somewhere in CIO).

6. When CIO is finished, it returns control to BASIC.

Wasn't that fun? For even more fun, try to trace what happens if interrupts occur during any or all of the above steps.

More Space

But why do we go through all this? Because, even though Atari saw fit to include all this good memory bank selection capability, they provided no software to use it. So why not just forget the bank select and pretend we are running on an Atari 800 or 400? Because the net gain to you, the BASIC or ACTION! or Assembler or whatever user, is about 5,000 bytes of user space. Your programs can be 5K bytes bigger. Your spreadsheets can contain many more cells. You can edit more text.

Of course, some programs (such as VisiCalc) which do not use a standard DOS or which use a heavily protected disk (such as the Microsoft BASIC extensions) will not be able to take advantage of the extra memory. But they, too, can use these techniques to extend their capabilities if the software companies producing them will decide that the XL machines are worth the little extra effort.

C

EPSON*, NEC*, PROWRITER*, GEMINI*, OKIDATA 92*

NEW!
ATARI*



The only self-booting grafix handler for dumps in **horizontal** format — all mach. lang. — **Lister** incl. — all modes — mixed modes — change aspect ratios, etc. while running other programs — assem ed — basic or no cartridge — demos, utilities, fonts, included — dump, create forms, stationery, calendars, requires interface. **\$26.95**

diskwiz-II

Now for single/double density. Repair, explore, alter, duplicate, map, speedcheck, bad sector (810), block move, trace, special print capabilities, disassembler, new speed, new ease, new functions, special printing functions, excellent repair tool w/instr. — even better than before! The best repair/editor/duplicator at any price — still at the lowest price. (Updates avail. for a small fee.) **\$28.95**

1st Class Postage Paid

California Residents add 6%, Foreign Orders add \$2.50

C.O.D. add \$2.00 — No credit cards

Prices subject to change

(213) 376-4105

ALLEN

MACROWARE

P.O. Box 2205

Redondo Beach, CA 90278

* Indicates Trademark of non-related company

INSIGHT: Atari

Bill Wilkinson

Well, it's the new year and, even though I am writing this months before New Year's Eve, I'm going to make at least one resolution right now: I hereby resolve to write the articles which I have promised. (Except, of course, if I...naw, that's not fair. I'll even try to avoid those exceptions.)

So, in the spirit of that resolution, I'm going to deliver the fourth part of my series on writing self-relocatable machine language right now. This month. Immediately. After I feed you some tidbits first.

Keep Those Cards And Brickbats Coming

Recently, I have received several letters ("several" means more than four—I am seldom exactly deluged with mail) which all bear on one or two topics. Since there appears to be some interest in these two areas, I would like to touch on them this month. Normally, I acknowledge my readers by name when I answer letters. This time, however, several asked the same questions, and I am hesitant to single out any one letter. If you recognize, in this column, a response to a letter you wrote me, I offer my thanks for the ideas you have given me.

Machine Language

The questions about this topic ranged all the way from "How about a section for machine language beginners?" to "Are you planning any more about graphics from machine language?"

To begin, let me say that I do not intend to teach a tutorial machine language class through this column. A good tutorial would take about 200 magazine pages, minimum. That's about what *COMPUTE!* allots me for two years' writing. By the time the series were finished, I would hope that you would have been experienced programmers for over a year!

On the other hand, I will try to take the spirit of the questions to heart and include a little more material for those who are just beginning to learn machine language. (Unfortunately, that does not include this month's article, but I feel committed to finishing the series.)

Several of you have asked me if I will write

on how to do I/O and graphics from machine language. Unfortunately, I have already written a lot about these subjects (primarily from November 1981 to February 1982, but with many additions through the summer of 1982).

Alas, there is no beginner-level book which treats these subjects. Most of what I discussed in my articles is thoroughly explored in Atari's Technical Notes and Operating System documentation or *De Re Atari*, but you need to be well-versed in 6502 machine language before tackling either.

Probably the most popular books about the 6502 are those by Rodnay Zaks. My personal opinion is that they are good, but not great books. So, after you have digested Richard Mansfield's *Machine Language for Beginners* (*COMPUTE!* Books), you probably should be very careful about what book you pick up from your dealer's shelves. Pick one which appears appropriate to your level. But keep watching: More books are on their way.

The 1050 Disk Drive And DOS 3

I had promised that I would say no more on these topics, since there is obviously something of a conflict of interest for me here. (Atari hasn't bought our DOS XL, but some of the other disk drive manufacturers have.) But I have received several cogent questions and comments, and I will try to answer them as honestly as possible.

First, I heard from a couple of people that the 1050 drive does *not* support 32 sectors per track in its pseudo-double-density mode. The claim is that it only supports 26 sectors per track, a substantial reduction in capacity. Since I don't have a 1050 drive or the final version of DOS 3, I cannot directly verify or dispute this claim. (Is it possible that this claim is a result of an opinion which I myself expressed to a users group last spring?) I can only reiterate that it was an examination of a preliminary copy of DOS 3 which resulted in my comments.

The other letters I received either chided me for not giving more details on DOS 3 or simply asked whether it would work with...well, almost anything (Atari 810 drives, RAMDISKs, Mosaic boards, etc.). First, let me state that I have not

been able to exhaustively test DOS 3. The preliminary version works on an Atari 800 with an 810 drive. Beyond that, I cannot say.

DOS 3 achieves its random access file capability by segmenting the disk into 128 blocks of 1K each. Obviously, with so few blocks, one can keep a pointer to each block in memory at all times. In fact, the VTOC (which is a bitmap on DOS 2) is also the file block map (which doesn't exist on DOS 2, hence no random files), all nicely packed into only 128 bytes of your computer's memory per drive.

And that is beginning to get more technical than I meant to get in this section, but let me close by noting that expanding this scheme to a 5 megabyte drive would imply either 40,000 bytes per block on the disk (and remember, a block is the smallest possible file size) or 10,000 bytes of VTOC and file map per drive in your main memory (in order to maintain the 1K block size). And that is why I said in my previous articles that DOS 3 does not expand well.

Anyway, I found it surprising that Atari would introduce the double-sided drives of the 1450XLD with DOS 3. But maybe I'm going to be surprised again.

A Tidbit

Once again, I am indebted to Steve Lawrow, the author of our MAC/65 assembler for telling me of another discovery about Atari BASIC which I shall share with you.

I have been traveling around demonstrating our new BASIC XL to various user groups; and, quite naturally, I have found several quick and easy programs which show off the language. One of my favorites is the following little gem:

```
1 REM
2 REM
3 REM
...
99 REM
100 POKE 20,0
101 IF I<200 THEN I=I+1 : GOTO 101
102 PRINT PEEK(20)
```

The object of this little gem is to get a bunch of do-nothing lines (in fact, 99 REMarks) in a program and then see how much they slow down the loop in line 101. Location 20 is the $\frac{1}{60}$ second clock tick ($\frac{1}{50}$ second in countries using 50Hz power systems), so the result of lines 100 and 102 is to print out the elapsed time in clock ticks.

Well, I usually run this programette in Atari BASIC first, Atari Microsoft BASIC second, BASIC XL in slow mode third, and BASIC XL in FAST mode last. (See the chart for timings.)

Steve mentioned to me, though, that the timings of Atari BASIC (and slow-mode BASIC XL) were dependent on the line numbers chosen.

Skeptically, I renumbered the program to look like this:

```
1 REM
2 REM
3 REM
...
99 REM
4000 POKE 20,0
5000 IF I<200 THEN I=I+1 : GOTO 5000
6000 PRINT PEEK(20)
```

Sure enough, all the BASICs (except, naturally, BASIC XL in FAST mode) speeded up a little (again, see the chart). Why?

I know the answer for Atari BASIC and BASIC XL, and I suspect it is the same answer for Microsoft BASIC. When these BASICs need to make a line number search, they place the line number being searched for in a particular memory location. Then they search through the program, a line at a time, looking for a match on the numbers. As they search, though, they always check the high bytes of the line numbers first. If the high bytes do not match, they don't bother to check the low bytes.

In our first example, since all the line numbers were less than 256, all the high bytes were the same, so the search took slightly longer. In the second example, though, the GOTO statement caused a search for line number 5000, whose high byte is *never* the same as those of any of the other lines. Bingo, fast search speed.

What does this mean? When writing in BASIC, it might be a good idea to modify the old traditional line-numbering-by-10. Purposely break your program up into sections so that the target lines of GOTOs and GOSUBs all differ by at least about 300, and you will help BASIC do its searching a bit faster. (And even though BASIC XL in FAST mode is not affected by these foibles when working with absolute line numbers, even it will be helped when in slow mode or when you use variable names or expressions as GOTO/GOSUB targets.)

And, incidentally, you might remember that the Microsoft version of this program must be typed exactly as shown to get these timings. Using longer variable names, more spaces in a line, more variable names, etc., will significantly slow down Microsoft BASIC. Often to the point where it is slower than Atari BASIC.

Anyway, here is the chart of timings. The Microsoft BASIC Integer version timings were obtained by appending a % to all variable and constant usage in lines 101 and 5000. Try some timings like this yourself. You'll be amazed.

Timings For The 99 REM Benchmark

	Atari BASIC	Microsoft Fltg Pt.	Microsoft Integer	BASIC XL Slow	BASIC XL Fast
GOTO 101	178	169	155	125	35
GOTO 5000	162	160	145	110	35

Self-Relocatable Machine Language, Part 4 (At Last)

Since it has been three months since Part 3 of this sort-of series appeared (COMPUTE!, September 1983), let me briefly summarize why self-relocatable machine language (ML) is desirable:

1. If all your ML is self-relocatable, you can load as many (or as few) modules as desired without worrying about where to put them in memory.
2. If you are using ML within Atari BASIC strings, remember that the strings can be moved by BASIC, so the ML virtually *has* to be self-relocatable.
3. Various pieces of systems software (for example, Atari BASIC, Pascal, Microsoft BASIC, some compilers) insist on using certain portions of memory. Since the pieces they insist on are not consistently the same, it is an advantage to be able to load your ML (especially device drivers, utilities, etc.) wherever the systems software leaves you a hole.

Also, let me summarize some of the rules for "Safe Relocatable Techniques," as presented in September:

1. Change JMPs to branches.
2. Save register values in the stack, not in fixed memory.
3. From BASIC, pass the address of a string as a location (or series of locations) to load from or store to. Note that Part 3 discussed how the ML string itself could be used for this purpose.
4. Move ML from relocatable memory to fixed memory temporarily.
5. Avoid load, store, and transfer instructions which refer to locations within your own module.

Finally, let me remind you that I promised to tell you how to utilize more than 255 bytes of relocatable storage and how to generate pointers to such storage without the "benefit" of help from a calling BASIC program. I shall attempt to fulfill my promise.

The techniques I will discuss here require a very small segment of nonrelocatable ML as well as one or (better) several zero page pointers. If you are in really dire straits, you can make do with temporary locations for both those requirements, but if possible you should find a way to preserve the required memory exclusively for your routine. In fact, the rest of this discussion assumes that you *have* managed to preserve the locations.

First Requirement: Find Yourself

You must have a subroutine, located at a fixed location, which looks like this:

```
BASE      =      $CE
          *=      $680
FINDME
    PLA
    STA     BASE+1
    PLA
    STA     BASE
    PHA
    LDA     BASE+1
    PHA
    RTS
```

Note that I have placed this routine in the infamous Page 6 and have used a fixed zero page location. These choices are for convenience, for illustration. Feel free to make your own choice of locations.

And just what does this routine *do*? How does it work? Quite simply, it finds the address of the program which called it. More precisely, it finds the address of the last byte of the three-byte JSR instruction with which a relocatable program calls it. An illustration of the calling program will help:

```
          *=      ???
START
    JSR     FINDME
BASEPT1   =      *-1
    ...
    LDY     #DATABYTE-BASEPT1
    LDA     (BASE),Y
    ...
DATABYTE .BYTE 99
```

Do you follow this? When FINDME is called via the JSR, it places the address of BASEPT1 into the zero page location called BASE. Then the Y register is loaded with the offset from BASEPT1 to DATABYTE and used an index for the LDA instruction. (This is similar to the technique discussed in Part 3, but it could only be used from BASIC USR calls.)

The limitation of this technique is that the data location (for example, DATABYTE above) must be located no more than 255 bytes away from the JSR (for example, BASEPT1). If you are writing a package of several small routines, this may not prove to be a limitation. After all, each routine could call FINDME if needed, and each routine could thus have its own storage areas, located no more than 255 bytes from the respective call to FINDME. If you are writing a subroutine library or a device driver, this might prove to be a very worthwhile option.

Note the side "benefit" to the scheme: If you call FINDME each time you enter a routine, then BASE may prove to be a really very temporary location and can be shared with other routines.

So far, so good. But suppose that you really

do need a large data area or program, all self-relocatable. Well, then, your program might have to do this:

```

* =      ???
DATABASE = $CC

START
JSR      FINDME
BASEPT1  = *-1
OFFSET1  = DATABYTES-BASEPT1

...
CLC
LDA      BASE
ADC      #OFFSET1&255
STA      DATABASE
LDA      BASE+1
ADC      #OFFSET1/256
STA      DATABASE+1

...
LDY      <some offset in DATABYTES>
LDA      (DATABASE),Y

...
DATABYTES .BYTE 1,2,3,4,5,6,7,8,9,10

```

Even more confused? You have a right to be. Here, we actually develop the base address of a data area and place it in a new zero page location. Now we can access the data area from anywhere in our self-relocatable ML by simply placing an offset within that data area into the Y register. Again, this limits the size of access to 256 bytes (the range of values the Y register can take on), but now the program can be as large as desired.

Finally, what happens if you actually do have a data area larger than 256 bytes? There are several possible solutions, none of them easy. If no "array" within the data area is larger than 256 bytes, you could simply develop several zero page pointers—one for each group of 256 bytes or less—using the ADC #OFFSET technique presented above.

If you have a single array or table which is larger than 256 bytes, the chances are that you have already developed some method of addressing into it (since the 6502 limits you to index sizes of 0 through 255, unless you play with indirect-Y addressing and calculated zero page pointer values). You need only use the contents of DATABASE, as generated above, in place of an absolute address for the start of the array or table, and your address calculations will be similar or even identical.

If you are lost at this point, don't worry. Much of what I just said will suddenly be meaningful as you write more and more advanced machine language programs. Just keep this article for handy reference.

Second Requirement: Calling Yourself

Suppose you want to call subroutines within your self-relocatable ML. How do you do it?

Of course, if the subroutine is at a fixed location (in ROM somewhere), you need do nothing special. The JSR instruction insists on an absolute

address, and you simply supply one. But what happens if the routine you want to call is itself part of the self-relocatable ML?

Advice: Avoid doing what I am about to describe if you possibly can. However, if you need to write ML which *must* use these techniques, read on.

First, you could simply write some self-modifying ML. An example:

```

START
BASEPT  JSR      FINDME
ROUTINE =      *-1
CALL1   =      ROUTINE - BASEPT
        =      CALL + 1 - BASEPT

...
LDY      #CALL1
CLC
LDA      BASE
ADC      #SUB1&255
STA      (BASE),Y
INY
LDA      BASE+1
ADC      #SUB1/256
STA      (BASE),Y

...
CALL     JSR      0; ADDRESS WILL BE GENERATED
        ...
ROUTINE  ...
        RTS

```

Simply, did he say? Well, it's not as bad as it looks. After all, if we could generate the address of a table and place it in zero page, why can't we place a subroutine's address directly into our ML? Of course, we must do the placing indirectly, since even the address of the JSR instruction is self-relocatable. Did you note that CALL1 is an offset to the first address byte in the instruction? It wouldn't do to modify the instruction byte!

Another way of doing JSRs like this might be to place yet another small routine in nonrelocatable memory. You could (1) load the A and X registers with the offset to the desired subroutine, then (2) JSR to the nonrelocatable routine which would calculate the actual address you desired, and (3) JMP to that location. When the subroutine returned, execution would continue at the instruction after your JSR. ©

Use the card
in the back
of this magazine
to order your
COMPUTE! Books

INSIGHT: Atari

Bill Wilkinson

This month we'll begin to explore some of the techniques involved in creating a general-purpose formatted screen I/O routine in BASIC. "And just what is a general-purpose formatted screen I/O routine?" you quite rightfully ask.

A "New" Kind Of Screen Editor

Briefly, what I am trying to do is produce a method whereby the programmer may specify certain areas of the screen as "label" or "title" areas, which may not be modified by the user. Other parts of the screen then become the Input/Output (I/O) areas. The user will not be able to change any part of the screen except the designated I/O areas, but he or she will be able to "randomly" access any area and change it. When the screen is filled in properly, the user pushes a single key (I intend to use ESCape) and the screen is automatically read into data variables in memory, where the program may process them or write them to disk.

The concept is certainly nothing new. Main-frame installations such as airline reservation systems have been doing exactly this for years. And I am sure that programs already exist for the Atari computers which work in a like fashion. So why am I writing these routines? For practical use here at OSS. Believe it or not, we intend to have a sales order entry system, complete with accounts receivable and general ledger interface, up and running on an Atari computer.

Surprised? Didn't think the Atari was capable of such sophisticated work? Truthfully, as the machine is shipped from Atari, it is not. The big missing link is large amounts of disk storage. We intend to use at least two double-density, double-sided drives (or equivalents), and may find that we need three or four.

And why are we doing this on an Atari computer, instead of a CP/M or MS DOS machine? Quite frankly, because we have the equipment already paid for and because we have yet to see an adequate order entry system even for such

"bigger" machines.

Anyway, so far I have written three of the workhorse subroutines of my formatted screen routines: (1) Display fixed information at fixed locations on the screen, (2) Display variable information (presumably obtained from a disk file) on the screen, (3) Edit the variable information (or enter new information).

Routine number three is both too big and too complicated to put in this month's column. Also, it runs fine in BASIC XL; but when I tried to translate it to Atari BASIC, it got bigger and slower and may not be too usable. If there is enough interest, I might be persuaded to write about it in a future column. Routines 1 and 2, though, are so surprisingly small, simple, and elegant when written in Atari BASIC that I felt you would enjoy seeing them. So let's look at them before explaining how they work.

Routine 1: Fixed Setup

```
30000 REM set up fixed screen areas
30010 TRAP 30020 : DIM DATA$(50)
30020 TRAP 40000 : RESTORE PTRDATA
30030 READ DATA$ : IF DATA$="*" THEN
      RETURN
30040 POSITION VAL(DATA$(1,2)),VAL(D
ATA$(3,4))
30050 PRINT DATA$(5); : GOTO 30030
```

Routine 2: Variable Display

```
31000 REM display variable data area
31010 TRAP 31020 : DIM DATA$(50)
31020 TRAP 40000 : RESTORE PTRDATA :
      QPTR=1
31030 READ DATA$ : IF DATA$="*" THEN
      RETURN
31040 POSITION VAL(DATA$(1,2)),VAL(D
ATA$(3,4))
31050 PRINT SCREEN$(QPTR,QPTR-1+VAL(
DATA$(5,6)));
31060 QPTR=QPTR+VAL(DATA$(5,6)) : GO
      TO 31030
```

Listing 3: A Tester For The Routines

```
100 DIM SCREEN$(200)
110 SCREEN$="ZUCKERMAN 95099C"
```

```

200 REM fixed data
210 DATA 0810Name:
220 DATA 0412Zip Code:
230 DATA 0816Code:
240 DATA *
300 REM variable data parameters
310 DATA 151010
320 DATA 151205
330 DATA 151601
340 DATA *
400 REM the actual test program
410 GRAPHICS 0
420 PTRDATA = 200 : GOSUB 30000
430 PTRDATA = 300 : GOSUB 31000
440 REM just loop here for now
450 GOTO 440

```

Even though I have presented this example as three separate listings, if you would like to see its effects, you should type all the lines into a single program.

Addressable DATA

So, what's the secret of this simple yet (according to me) elegant program? Surprisingly enough, I find myself returning to the concept I explored in my very first COMPUTE! column (September 1981, for you "regulars"): addressable DATA statements. Very few BASICs have addressable DATA statements, yet when I look at this program I cannot understand why they don't.

The lines to look at carefully are 30020 and 31020, where the program says "RESTORE PTRDATA". When either of these routines is called, it expects that the variable PTRDATA will contain the line number of the beginning of some DATA statements which it must begin processing. So let's look at those DATA statements first.

In lines 210 through 230, we define the fixed fields on the screen as starting at a particular horizontal (X) position (the first two digits) and a particular vertical (Y) position (the next two digits). Notice how line 30040 reflects this usage with the VAL functions it uses in conjunction with the POSITION statement.

Similarly, in lines 310 through 330, the definitions of the variable fields are expressed as horizontal position (first two digits), vertical position (next two digits), and field length (last two digits). Again, lines 31040 through 31060 reflect these usages via VAL functions.

If you are wondering why I am making such a fuss over these two little routines, especially when it takes so much programming to prepare to use them, you probably haven't typed in the program to see what it does. Or, to be fair, you haven't seen the best part of all, the onscreen editor that's too big for this month's column.

INPUT Weaknesses

And why am I going to this much trouble, when I could use PRINTs and INPUTs to do the same

thing? Two reasons: (1) If I use PRINT and INPUT, I have to write the entire code each time in a form which makes my programs hard to read and understand. (2) The INPUT statement as implemented on most BASICs is a disaster, and Atari BASIC is no exception. There is no way, when using INPUT, to keep the user from hitting screen-editing keys or from entering too much or too little data.

Did I mention that the screen editing routine I have written allows the programmer to specify, via simple DATA statements, not only where and how big the variable data fields are on the screen but also what attributes they may have (for example, numeric, alphabetic, dollars and cents, etc.)? I didn't? Are you more interested now? Next month we'll continue our examination of screen I/O by making test runs of the example programs. C

COMPUTE!

The Resource.



Copy Atari 400/800 Cartridges to Disk
and run them from a Menu

ATARI CARTRIDGE-TO-DISK COPY SYSTEM \$69⁹⁵

Supercart lets you copy ANY cartridge for the Atari 400/800 to diskette, and thereafter run it from your disk drive. Enjoy the convenience of selecting your favorite games from a "menu screen" rather than swapping cartridges in and out of your computer. Each cartridge copied by Supercart functions exactly like the original... self-booting, etc.

Supercart includes: COPY ROUTINE - Dumps the contents of the cartridge to a diskette (up to 9 cartridges will fit on one disk.)

MENU ROUTINE - Auto loading menu prompts user for a ONE keystroke selection of any cartridge on the disk.

CARTRIDGE - "Tricks" the computer into thinking that the original "protected" cartridge has been inserted.

To date there have been no problems duplicating and running all of the protected cartridges that we know of. However, FRONTRUNNER cannot guarantee the operation of all future cartridges. Supercart is user-friendly and simple to use. **PIRATES TAKE NOTE: SUPERCART is not intended for illegal copying and/or distribution of copyrighted software. Sorry!!!**

SYSTEM REQUIREMENTS:

Atari 400 or 800 Computer / 48K Memory / One Disk Drive
Available at your computer store or direct from FRONTRUNNER. DEALER INQUIRIES ENCOURAGED
TOLL FREE ORDER LINE: (24 Hrs.) 1-800-648-4700 or for questions Call: (702) 786-4600
Personal checks allow 2-3 weeks to clear. M/C and VISA accepted.

Include \$3.50 (\$7.50 Foreign orders) for shipping.

FRONTRUNNER COMPUTER INDUSTRIES

316 California Ave., Suite #712, Reno, Nevada 89509 - (702) 786-4600

Others Make Claims... SUPERCART makes copies!!!

ATARI is a trademark of Warner Communications, Inc.

VIC20 /COM 64/ ATARI 400/800 SOFTWARE RENTAL CLUB

- RENT SOFTWARE for up to a month for 10% of the list price (20% for cassettes and disks) with option to purchase
- Call us for Atari and Commodore 64 Hardware Supplies
- Membership \$25/year with \$10 Renewal fee
- VISA/MC accepted



VIDEO HOME LIBRARY

RT. 3 BOX 309A52

CLINTON, TN 37716

(615)457-5068, 482-3893

Software orders add \$1.50 for shipping and handling.

INSIGHT: Atari

Bill Wilkinson

In this column, we continue the discussion of formatted screen techniques.

PUT And GET And The Text Screen

This is another one of those "Did you know?" tidbits. Did you know that when you use GRAPHICS 0 from Atari BASIC you have automatically opened the screen for GETting and PUTting via file number 6? It's true, and it is because Atari BASIC does not check the mode number for the GRAPHICS statement.

GRAPHICS 0 is thus exactly equivalent to:

```
OPEN #6, 12+16,0, "S:"
```

So if you need to GET or PUT from or to the screen, you can do it directly to file #6 without any further ado.

Unfortunately, there are a few gotchas involved in using GET and PUT to the Atari Screen graphics driver ("S:"), some of which you may have seen before, so let's discuss them, as well as ways around them.

The first problem is that if you use PUT #6 combined with POSITION statements or PRINT statements, you will probably end up leaving some inverse video spaces (white boxes) around on the screen, as Program 1 illustrates. This is because the screen graphics driver works almost (but not quite) like the screen editor driver ("E:", the normal channel #0 device which PRINT and INPUT use). Unfortunately, "S:" can't seem to handle its cursor properly, so it may be best to avoid using PUT #6.

Program 1: Problems With PUT #6

```
10 GRAPHICS 0
20 POSITION 30*RND(0), 20*RND(0)
30 PUT #6, 65+20*RND(0)
40 GOTO 20
```

How can we avoid PUT #6 if we have something we need on the screen? Simple. Use PUT #0 (if you have BASIC XL or any other product which allows PUT to file #0) or PRINT. If you use PRINT, of course, you will have to use

```
PRINT CHR$(X);
```

in place of PUT #0,X. And why does outputting to file #0 work where using #6 does not? Because #0 is opened to "E:", and there are several subtle differences between "E:" and "S:" where cursor

positioning and character I/O are concerned.

Unfortunately, while the problems with PUT #6 are fairly easy to get around, the problems with GET #6 must be dealt with directly. And why can't we simply use GET #0 in place of #6 here, as we did with PUT? Because, when you ask "E:" (channel #0) for a character, it waits until the user actually types in an entire line—terminated by a RETURN character—before returning anything at all to its caller (you are the caller via BASIC in this case).

The whole reason for using GET #6 is to allow ourselves to read individual characters from the screen. We simply can't use GET #0 or anything else which accesses "E:".

But this is putting the cart before the horse a little. Before "fixing" the problem, let's illustrate it with Program 2.

Program 2: Problems With GET #6

```
10 GRAPHICS 0
20 PRINT "ABCDEFGH IJKLMNOP"
30 FOR I=2 TO 12 : POSITION I,0
40 GET #6, CHAR
50 POSITION 20,20 : PRINT CHAR
60 FOR J=1 TO 200 : NEXT J : REM jus
  t a delay loop
70 NEXT I
```

I hope you actually stopped while reading to try out that listing. Bizarre, isn't it? It seems that you can't GET data from the screen without destroying it. Now, most of the articles which I have seen which note this problem suggest that the only safe fix is the following:

1. POSITION yourself on the character you want.
2. GET the character to a variable.
3. POSITION yourself again to the same location.
4. PRINT the character back onto the screen.

That fix will indeed work, but I would propose that an alternate solution is to simply print a "left arrow" (backspace) and then the character, thus avoiding the extra POSITION statement. In Program 2, we could simply add this line to fix things up:

```
45 PRINT CHR$(30);CHR$(CHAR);
```

Now that you know how to properly PUT and GET to the screen, you probably have a fair idea of how I built my onscreen editor. It isn't too hard to do anything you want to the GRAPHICS

0 screen, once you get past the quirks in the Atari OS.

Fettering Your NEXT

Probably every BASIC book you have ever seen tells you to properly nest FOR/NEXT loops. Aside from the neatness of it, there are some good and practical reasons. Consider Program 3.

Program 3: Obviously Invalid Nesting

```
10 FOR I=1 TO 6
20   FOR J=1 TO 3
30   NEXT I
40   NEXT J
```

Very few of you would deliberately write a BASIC program which looked like that. Even with the indentation I have given it, it should be obvious that something is wrong.

And, yet, it is fairly easy to write a program which will look proper and yet have the effect of that listing! Don't believe it? Try Program 4.

Program 4: A Subtle Problem

```
100 REM Program task: Print all numbers from 1 to 9, in a nested loop fashion. When the first sum of 15 or
101 REM greater is found, cease the operation. When the sum is 10 or more, don't print the result.
102 REM Repeat for the products of the same numbers in the same fashion.
110 print "I","J","SUM"
120 FOR I=1 TO 9
130   FOR J=1 TO 9
140     SUM = I+J
150     IF SUM > 14 THEN 200
160     IF SUM > 10 THEN 190
170     PRINT I,J,SUM
180   NEXT J
190 NEXT I
200 PRINT "I","J","PRODUCT"
210 FOR J=1 TO 9
220   FOR I=1 TO 9
230     PROD = I*J
240     IF PROD > 14 THEN 290
250     IF PROD > 10 THEN 280
260     PRINT J,I,PROD
270   NEXT I
280 NEXT J
290 END
```

Now this looks perfectly harmless, if somewhat pointless, right? It looks like it should work fine. Yet, if you will type it in and RUN it, you will find that line 280 will give you a NEXT WITHOUT MATCHING FOR error the first time it is reached. How? Surely line 210 is the FOR which matches the NEXT of line 280.

The Interpreter's Dilemma

If Atari BASIC were a compiler language, it would probably execute that program correctly. However, since it is an interpreter, it must work within the strictures of that mode. Interpreters, by their

very nature, cannot easily keep a history of all NEXT usages. It is enough that they remember where the FOR statements are, so that when a NEXT is encountered they can go back to the FOR to execute the loop another time.

Consider, then, the dilemma of the poor interpreter in the above program. In line 160, we are asking it to bypass the end of the inner FOR loop (since we know we are done with the previous usage of it) and start the next iteration of the outer loop (NEXT I). But wait. There is still a FOR J on the runtime stack, yet we are executing a NEXT I. What can we do?

Atari BASIC does what most modern "smart" BASICs do. If it finds a loop variable NEXT which does not match the last FOR on the stack, it presumes that the user has jumped out of the inner loop (as indeed we have here) since that is a common occurrence. So BASIC looks backward in the stack for a matching FOR. Eureka! It finds the FOR I only one level down in the stack, without any intervening GOSUBs, so its supposition seems confirmed. All works well.

However, look at line 150, wherein we jump out of all the loops. What have we left on the runtime stack now? Obviously, both a FOR I and a FOR J. Well, no real problem. After all, we know we jumped all the way out of the loop, don't we? We don't. Why not? Because a BASIC interpreter must presume that the BASIC programmer knows what he or she is doing. It is, unfortunately, perfectly legal to jump in and out of a loop in Atari BASIC. It is, in fact, even legal to have more than one NEXT for any given FOR.

So what can BASIC think when it gets to line 210 but that it is starting the inner FOR loop over again? It leaves the FOR I in place (for all it knows, the next statement it encounters might be a NEXT I) and adds a new FOR J.

Disaster really strikes in line 220. Poor BASIC is trying its best. Knowing that it is not uncommon for BASIC programmers to jump out of loops or to jump to the beginning of a loop to start it again, BASIC almost *has* to presume that the FOR I of line 220 is the beginning of a new outer loop. Besides, it already has a FOR I on its runtime stack. How can it allow another?

Well, if this is the beginning of a new outer loop, better throw away the old outer loop and any of its inner loops. Say good-bye to the old FOR I and FOR J; we're ready for another outer loop with a new FOR I. Right?

Wrong. But BASIC doesn't know about it while it stays in the FOR I loop, since it encounters no other FORs or NEXTs. In fact, the entire loop executes nicely with no problems, and the FOR is properly removed from the stack when the last value of I is reached. Did you notice that the stack is now empty?

Where did this NEXT J come from? FOR J was an inner loop and was thrown away when the outer loop was restarted.

The Fix In Atari BASIC

Actually, Atari BASIC is not a culprit here. Virtually every BASIC will have this same problem unless it makes a pre-pass through the user's program to detect possible inconsistencies (such as jumping out of nested loops). In point of fact, Atari BASIC is almost a good guy here. Recognizing that even with the best interpretation we could do, we could not prevent users from writing (or needing to write) structures such as I have shown you, we designed a "fix" into Atari BASIC.

The fix takes the form of the POP statement. POP simply removes the last level of the runtime stack. In Program 4, the easiest fix is

```
150 IF SUM > 9 THEN POP : POP : GOTO 200
```

(and a similar fix is needed in line 240, of course).

Notice I said that was the easiest fix. POP is usually not the best fix. Generally, you can write good and properly structured programs, with properly terminating FOR loops, without ever resorting to such extreme measures as the POP statement. Still, it is comforting to know that POP is around. Personally, I tend to use it whenever an error condition occurs and I want to get all the way back out to (for example) the menu level without leaving nasty GOSUBs or FORs on the runtime stack.

A curiosity: Did you notice that if the nesting in lines 200 through 290 is reversed (that is, if the FOR I occurs before the FOR J), the program will work correctly? Do you see why? Fundamentally, because you are now doing what BASIC expected you to do. Go try this example both ways on a Commodore or Radio Shack or whatever computer. Does either method work? I'd be interested in knowing.

If you ever get a NEXT WITHOUT MATCHING FOR error, look for this kind of structure in your program. If you find it, you can fix it with POP, but wouldn't it be nicer to write the program correctly?

A footnote to all of that: Can you begin to get an appreciation of what language designers must contend with? It is not enough that a language do what it is expected to do. A good language will come halfway toward helping its users over the rough spots.

Reading Object Code Files

Here's a loader for binary object files which will place them in memory at the location they were assembled for. The routine is written entirely in Atari BASIC, so it is slow. Next month, we'll present the same routine written in machine language, perhaps even in a version callable from a

BASIC program (just to speed things up).

Atari object files have a fixed and reasonable format. The first two bytes of the file are always \$FF and \$FF (255 and 255, in decimal). They serve as a check that the file is indeed an object file. The next two bytes are the starting address in memory of the first (and perhaps only) "segment," while the following two bytes are the ending address of the segment. These header bytes are followed by enough object bytes to fill up the memory from the starting address through and including the ending address.

If a file has multiple segments, each segment may or may not (programmer's option) be preceded by the same \$FF and \$FF bytes. Each segment must always be headed by both a start and an end address. Without further ado, then, the loader program, Program 5.

Program 5: Load A Binary Object File

```
100 REM binary object file loader
110 DIM NAME$(30)
120 PRINT "WHAT FILE TO LOAD ";
130 INPUT NAME$
140 OPEN #1,4,0,NAME$
200 REM get and check header
210 TRAP 400
220 GET #1,LOW : GET #1,HIGH
230 TRAP 40000
240 IF LOW=255 AND HIGH=255 THEN GET
    #1,LOW : GET #1,HIGH
250 START = LOW + 256*HIGH
260 GET #1,LOW : GET #1,HIGH
270 QUIT = LOW + 256*HIGH
300 REM read in a segment
310 FOR ADDR = START TO QUIT
320 GET #1,BYTE
330 POKE ADDR,BYTE
340 NEXT ADDR
350 GOTO 200 : REM try for another segment
400 REM trapped to here, assume end-of-file
410 CLOSE #1
```

Since I'm running out of time and space this month, I will let the explanation of object file format, above, serve for now as an explanation of this program. I will warn you, however, that I cheated a bit in line 240 to make the multiple segment loading easier. The routine will try to load *anything* into memory, whether or not it is truly a binary object file. If your memory dies a violent death (fixable only by turning power off and back on), you tried to load something other than an object file with this. Naughty.

Next month some notes on destination strings in Atari BASIC. And maybe—just maybe—we'll play around with Atari screen I/O a little more. **C**

COMPUTE!
The Resource

INSIGHT: Atari

Bill Wilkinson

Well, here it is April again. Those of you who read my April column last year may recall that I devoted much of my space to an April Fool announcement of the "new Atari COBOL." Would it surprise you to learn that this year I will also "announce" some new products from Atari?

We're going to be exploring some medium hefty programming logic (in machine language) next month: How to use your 1050 disk drive in enhanced density with Atari DOS 2.0s. For now, though, let's plunge into some wild speculations, rumors, and April fun.

Incredible Integration

Since last year's April announcement was about some literally unbelievable software, it seems only fitting that this year we make a hardware announcement that's almost as doubt-provoking.

By the time you read this article, Atari will be shipping at least the first two of three magnificent new machines. These machines, while maintaining almost full compatibility with existing Atari hardware and software, add the full power of an intelligent peripheral expansion bus. Imagine an Atari computer hooked up to a 5- or 10-megabyte hard disk drive, a true parallel printer interface, a high-speed modem, and maybe even a CP/M emulation package.

I mean, we're talking about possibly moving data to and from a disk drive at 30,000 to 60,000 bytes per second! Imagine taking less than two seconds to load the largest possible programs. And perhaps talking via a serial interface (or, better yet, a local network) to one or more other computers at the same time—at data rates perhaps three to ten times what an 850 Interface Module is capable of.

Software Compatibility, Too

Of course, if you are a realist, you will say, "Okay for the hardware. But what about software and software compatibility?" Would you settle for a smart peripheral bus that intercepts the OS (CIO) if you are trying to do I/O via the old serial bus. You know, the cable that links your 400/800/1200 to the disk, printer, etc.? Could you accept the fact that it checks to see if you have (for example) moved "D1:" to your hard disk drive and sends

the request there instead of to your 810? All automatically?

Not enough? How about if these new machines even provided ways for third-party hardware vendors to add their own boards and automatically link in device handlers for them? Imagine a music synthesizer accessed as simply "M:", thus easily callable from even Atari BASIC. Could *any* computer manufacturer possibly design such a well-integrated system?

How about if the new machines even came with a faster math package, so that they were the fastest computers in the home computer marketplace? (I choose to define a home computer as one which costs less than \$1000, including at least a disk drive.)

April Fool

Well, you knew it couldn't last, didn't you? Sigh. But it was nice to dream for a paragraph or two, wasn't it? Now, are you ready for the bitter reality?

Surprise! This is my April Fool gag for this year: Almost everything you just read about the new machines is the absolute truth. Honest. No gag.

In fact, as I write this article in January, the machines I have described to you are arriving in stores by the truckfuls. And why, you ask, haven't you seen these wonder computers advertised? Ah, but you have. They are called the 600XL and the 800XL (with big brother 1450XLD still to come). But if all I claimed is true, why hasn't Atari proclaimed it to the computer world as the greatest advance ever in home computers? Now *there* is the April Fool question.

If Atari can solve some advertising and delivery problems, I think you will see a wealth of capabilities added to the new XL machines second only to the selection available to Apple II owners.

Oh, yes. I did throw one April Fool joke into the description above. Can you guess what it is?

Final Foolishness

The descriptions of the Atari super machines were accurate except for one April Fool joke. Sigh. Unfortunately, the part about the advanced, fast BASIC being built-in is still just a dream.

INSIGHT: Atari

Bill Wilkinson

Learning How

A month or two ago, I stated that I couldn't possibly teach beginning machine language programming in this column—it would consume my entire output for a year or more. And yet I continue to get letters that ask me "How do you learn to write programs?"

I believe that those who ask the question are not asking for a tutorial on the foibles and pitfalls of the FOR-NEXT loop. Nor are they really asking about the intricacies of the 6502 instruction set. Most of them have already mastered the tutorial-level material on their chosen language. What these perplexed people are really asking is "What good is all this programming stuff, anyway?"

And that is not really surprising. So many tutorials tell you *how* to write a program to do such and such. So few discuss *why*. Too often, learning to program is approached like learning a foreign language. Memorize the conjugations and punctuation; put sentences together like this; and if someone asks you "G'dye moya k'neega?" you know what to answer (providing you were studying Russian instead of Spanish).

Computer Conversations

But the need to learn human languages is obvious: The first time you feel hungry in Paris, you can ask for directions to a restaurant in your best Berlitz French. You don't have to "design" a conversation. Not so with learning to program: "Okay, now I know all these neat keywords and syntax and punctuation. How do I start a conversation?" Well, as I hinted above, the secret is that you must *design* a program.

To some, this design process is simple and obvious. Others never really get the hang of it. (Would it surprise you to learn that many professional programmers never become expert at designing? They make their living implementing other people's designs.) And many, like myself, become somewhat proficient at a few kinds of designs while remaining incompetent at others. (My lament: I don't think I will ever achieve the level of creativity necessary to design a really good game.)

Now, all the above philosophizing surely has some purpose, you hope. Indeed, I think it does.

Kibitzing

I have been promising for a few months now that I would provide patches to allow the Atari 1050 drive to work in enhanced mode with good old Atari DOS 2.0s. Well, I finally gathered enough information to begin the task, and I thought you might enjoy looking over my shoulder while I tackle the problem.

This will be a kind of short diary of what I have gone through. There have been more side-tracks and bugs and flat-out boo-boos than I can find room for here. And I won't even tell you how many assemblies I have made (though I will say I made about 10 or 12 just looking for the best of several possibilities for a series of shift instructions).

Even though I admire and strive for a "clean" design, I am apt to take the course of least resistance if I am confident it will work properly. With that in mind, then, let us begin tackling our task.

Note: I will make frequent reference to the listing of Atari DOS 2.0s as published in the book *Inside Atari DOS* from COMPUTE! Books. Page numbers and line numbers in square brackets [131: 1350] refer to the book.

It will *not* be necessary to own the book to understand most of what is going on, but having the book available will make it easier. Also, if you do not understand machine language, neither the book nor my explanations will be easy to follow, but you can still use the results (which will appear next month).

The 1050 And DOS 2.0s

The first thing we must always do is define the task. Here, that is deceptively simple to do: Make the enhanced density mode of the Atari 1050 drive work with Atari DOS 2.0s.

The next step is much harder: Design the implementation of the task. And, actually, this single step consists of many substeps. For example, let's first investigate the facts which I knew when I started.

The drives:

Item: An Atari 810 drive has 40 tracks of 18 sectors of 128 bytes each. That's a total of 720 sectors.

Item: An Atari 1050 drive has 40 tracks of 26 sectors of 128 bytes each, for a total of 1040 sectors.

Item: A 1050 will automatically read either density diskette (single or enhanced), but it formats a new diskette according to the format command it receives. In particular, a ! command (\$21) causes single-density formatting, while a " command (\$22) causes enhanced density.

The software:

Item: DOS 2 is capable of accessing both 810 drives and their double-density equivalents (drives with 40 tracks of 18 sectors of 256 bytes each).

Item: There is an inherent limit of 1024 sectors in DOS 2, since it allows only a 10-bit sector number in the link field of each sector. Also, on a single density diskette, DOS 2 accesses only 719 of the 720 sectors.

Item: The listing of Atari DOS. Actually, this is *not* a "known" item, and much of what follows is a discussion of what I learned and applied from reading the listing several times.

Finding The Format

Armed with these knowns, let's tackle the unknowns. It seemed to me that the first point to attack was the disparity between what the 1050 was capable of and what DOS 2 would request of it. All of a sudden, DOS 2 must be able to understand three different kinds of disk formats. Question: How can DOS tell what format a particular diskette is?

The answer is to be found in the DOS listing [66: 2213-2222]. During initialization, a status request is made of each drive. When the drive responds, one of the bytes it returns to the computer describes the drive's type. In particular, the listing makes it clear that a double-density disk has bit 5 (\$20) set on. DOS 2 uses this bit to differentiate between 128-byte and 256-byte sectors.

All very well, even assuming that an enhanced mode 1050 returns a zero bit here (which it does, thus properly indicating 128-byte sectors). But what distinguishes an enhanced density diskette? I confess that I obtained the answer to this question through a simple experiment: I simply booted a system with an Indus 1050-compatible drive as D2 and looked at the status value it returned during DOS initialization. Lo and behold, it returned \$80. Not surprisingly, the high bit is off in 810 and double-density modes. Voilà.

Sector Limits

The second major question to investigate is "How many of the 1050's sectors can we make DOS 2

utilize?" Well, we already know that 1024 is an upper limit. Is there any other limiting factor? The answer is in the layout of the Volume Table Of Contents (VTOC) under DOS 2. The VTOC contains a single bit for each accessible sector on the disk (a scheme known as a *bitmap*, though Atari literature often uses *VTOC* and *bitmap* interchangeably). If a bit is on (1), the corresponding sector is available. If a bit is off (0), the sector is in use. With eight bits per byte, then, there must be 90 bytes in the bitmap.

DOS 2 allows only a single sector (in this case, 128 bytes) for the VTOC of each diskette. While we could circumvent this restriction, it would require a lot of work, and might cause some secondary problems. (I don't want to go into this subject more now, but it cost me four to six hours of investigation before I decided against a two-sector VTOC.)

In 128 bytes, there are 1024 bits. So it would seem that the limit on number of sectors is indeed 1024. Alas, it is not to be. The description of the VTOC clearly calls out usages for the first six bytes (DOS type, maximum number of sectors, current number of sectors, write-required flag) and reserves the next four. So now we are down to 118 bytes and 944 sectors. Is that our limit?

A Final Of 976 Sectors

At first, I was inclined to say it is. But I pored over the listing a couple more times, checked every memory reference that was related, and finally concluded that we could use the four reserved bytes. Which gives us 122 bytes and a final maximum of 976 sectors. Well, that doesn't seem too bad. We are only 64 sectors away from the theoretical maximum and surely a lot better off than with a limit of 720 sectors.

So this is our plan: Use the upper bit (\$80) of the drive status to recognize an enhanced density diskette; allow 975 sectors (DOS 2 always throws away the first possible sector); displace the bitmap in the VTOC by 4 bytes on the low end and lengthen it to 122 bytes.

Implementing Our Plan

By the time I had decided on a plan, over half the time I had allotted to this project had elapsed. As I write this, all the allotted time is gone, and I am not done yet. Sounds like a typical software project. Anyway, this month I will tell you of the difficulties I faced. Next month we can decide how well I faced them. In any case, let's begin the next step.

Before I could start the actual coding of the modifications, I had to find all the places in DOS which would be affected by my scheme. While many parts of DOS are affected by a change in density (from 128- to 256-byte sectors), there are

only a few routines which actually care about such things as disk status, where the VTOC's bitmap is, and how many sectors are available.

Some of the routines I could successfully ignore. For example, when you delete a file and free up its sectors for later use, you must bump the count of free sectors. But if the rest of DOS is working, you don't have to check for validity of the bumped value. The same thing is true when we allocate a free sector and must decrement the count. And the boot process cares whether we are using 128- or 256-byte sectors, but it doesn't care how many sectors are on the disk.

Some Areas Need Patching

But there *are* several spots which definitely need attention, so let's discuss them now (next month we discuss the solutions).

1. In the BSIO (Basic Sector Input Output) routine, there is a check for a format command [65: 2144]. DOS 2 simply compares the current command with \$21 (!) and makes a decision according to an exact match. Now, though, we must allow for either \$21 or \$22 (") as format commands.

2. In DOS initialization [66: 2218], each accessible drive is checked for its status. DOS 2 ignores all bits of the status except bit 5 (\$20) and stores a 1 or 2 (single or double density) in the drive table (DRVTL) for each drive so checked. We need to find a way to capture and use bit 7 (\$80), preferably by keeping it in DRVTL, also. Fortunately, the only other routine which accesses DRVTL is SETUP, which we discuss below.

3. In XFORMAT [79: 3510], the actual format command is stored in the DCB (for use by BSIO, as above). We need to allow for either \$22 or \$21, while DOS 2 allows only \$21.

4. Also in XFORMAT [79: 3547, 3552], the maximum number of sectors and number of sectors available are stored in the VTOC which is being created (for the newly formatted disk). Currently, DOS 2 simply uses LDA # (load immediate value) to store what it thinks is the only possible count (707). We must provide for the enhanced density count as well.

5. Again in XFORMAT [80: 3559-3570], there are several assumptions made about how big the bitmap is and where the directory and boot sectors are to be represented in the map. Since we will move the base of the map down four bytes, we must provide for variable numbers here, as well.

6. In FRESECT [90: 5166], the base of the bitmap is assumed to be byte 10 (\$0A) of the VTOC. We must change the assumption.

7. In GETSECTOR [91: 5199, 5202, 5239], similar assumptions about the bitmap are coded via immediate loads.

8. In SETUP [92: 5288], which is called by

every major routine in DOS 2, the type byte stored in DRVTL (see item 2, above) is simply transferred to a global location (DRVTP) for use by other routines. If we change what is stored in DRVTL, we need to change how and what we store in DRVTP.

Keeping The Patches Small

And that's it. Not too bad, right? If only that were true. Remember, our goal here is to *patch* the standard version of DOS without affecting its normal operations and without requiring a reassembly of the whole thing to make our patches fit. In general, then, the smaller and fewer the patches, the better.

The real problem here is the number of load immediate instructions, used to implement what are now to become invalid assumptions. If these were three-byte instructions (such as loads from a non-zero page memory location), we would have a simple task: Change the values in the locations being loaded.

Since they are load immediate instructions, though, our only choices are to either make large and cumbersome patches (generally JSRs to subroutines which will do the work, but remember that JSR occupies three bytes), use loads from zero page (a neat alternative, but we have no zero page available to us), or to continue to use load immediate.

Self-Modifying Routines

My choice? Continue to use load immediate. But how? By producing some (shudder at this next phrase, please) self-modifying routines. Remember how I said at the beginning that I sometimes took the path of least resistance? This is one of those sometimes.

The "trick" which allows my scheme to work is relatively simple: Every routine which needs a load immediate changed is only used by DOS 2 after a call has been made to SETUP. Basically, SETUP examines the disk number and drive type and produces various pointers and values in fixed locations for use by other, higher-level routines. What would be more appropriate than for SETUP to also set up the needed values which will be loaded in immediate mode?

And this is, indeed, the plan I tried. At the point where SETUP stores the drive type [92: 5288], I placed a JSR to my patch-it routine. And my patch-it routine used the disk type information to determine which of a pair of immediate values would be used in each of the cases noted above. It looked like it would work.

Fitting The Patch Into DOS.SYS

Except (You knew that was coming, didn't you?) where do I put the patch? I have discussed this subject before, so let me succinctly say that the only sizable patch area in DOS.SYS is at location

\$1501, in the gap between DOS.SYS and Mini-DUP (the root of DUP.SYS). There are exactly 63 bytes available there. And my routine was about 85 bytes long.

The story of how I pared my patch down to fit (just barely) will have to wait for next month. Fortunately, it is a short patch. Also fortunately, there are a couple of small patch spaces still floating around in DOS.

Incidentally, if you were looking for the continuation of my notes on how to load saved binary files, keep looking. It turns out that the subject has direct bearing on what we are doing here, so it seemed not inappropriate to postpone it a month (or possibly two).

Use the handy
reader service cards
in the back of the
magazine for
information on
products advertised in
COMPUTE!

Program Your Own EPROMS

► VIC 20
C 64 **\$99.50**

PLUGS INTO USER PORT.
NOTHING ELSE NEEDED.
EASY TO USE. VERSATILE.

- Read or Program. One byte or 32K bytes!

OR Use like a disk drive. LOAD,
SAVE, GET, INPUT, PRINT, CMD,
OPEN, CLOSE—**EPROM FILES!**

Our software lets you use familiar BASIC commands to create, modify, scratch files on readily available EPROM chips. Adds a new dimension to your computing capability. Works with most ML Monitors too.

- Make Auto-Start Cartridges of your programs.
- The *promenade*™ C1 gives you 4 programming voltages, 2 EPROM supply voltages, 3 intelligent programming algorithms, 15 bit chip addressing, 3 LED's and NO switches. Your computer controls everything from software!
- Textool socket. Anti-static aluminum housing.
- EPROMS, cartridge PC boards, etc. at extra charge.

- Some EPROM types you can use with the *promenade*™
- | | | | | | |
|-------|-------|---------|-------|-------|---------|
| 2758 | 2532 | 462732P | 27128 | 5133 | X2816A* |
| 2516 | 2732 | 2564 | 27256 | 5143 | 52813* |
| 2716 | 27C32 | 2764 | 68764 | 2815* | 48016P* |
| 27C16 | 2732A | 27C64 | 68766 | 2816* | |

► *Commodore Business Machines

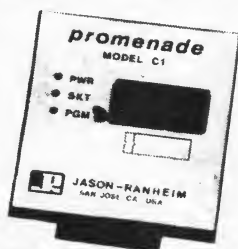
*Denotes electrically erasable types

Call Toll Free: 800-421-7731
In California: 800-421-7748



JASON-RANHEIM

580 Parrott St., San Jose, CA 95112



Commodore 64
and
VIC-20

SuperTerm

\$149⁹⁵

Telecommunications

with a difference!

Unexcelled communications power and compatibility, especially for professionals and serious computer users. Look us over; **SuperTerm** isn't just "another" terminal program. Like our famous Terminal-40, **it's the one others will be judged by.**

- **EMULATION**—Most popular terminal protocols: cursor addressing, clear, home, etc.
- **EDITING**—Full-screen editing of Receive Buffer
- **UP/DOWNLOAD FORMATS**—CBM, Xon-Xoff, ACK-NAK, CompuServe, etc.
- **FLEXIBILITY**—Select baud, duplex, parity, stopbits, etc. Even work off-line, then upload to system!
- **DISPLAY MODES**—40 column; 80/132 with side-scrolling
- **FUNCTION KEYS**—8 standard, 52 user-defined
- **BUFFERS**—Receive, Transmit, Program, and Screen
- **PRINTING**—Continuous printing with Smart ASCII interface and parallel printer; buffered printing otherwise
- **DISK SUPPORT**—Directory, Copy, Rename, Scratch

Options are selected by menus and EXEC file. Software on disk with special cartridge module. **Compatible with CBM and HES Automodems**; select ORIG/ANS mode, manual or autodial.

Write for the full story on SuperTerm; or, If you already want that difference, order today!

Requires: Commodore 64 or VIC-20, disk drive or Datasette, and compatible modem. VIC version requires 16K memory expansion. Please specify VIC or 64 when ordering.

Smart ASCII Plus . . . \$59⁹⁵

The only interface which supports streaming — sending characters simultaneously to the screen and printer — with SuperTerm.

Also great for use with your own programs or most application programs, i.e., word processors. **Print modes:** CBM Graphics (w/many dot-addr printers), TRANSLATE, DaisyTRANSLATE, CBM/True ASCII, and PIPELINE.

Complete with printer cable and manual. On disk or cassette.

VIC 20 and Commodore 64 are trademarks of Commodore Electronics, Ltd.

(816) 333-7200

Send for a free brochure.



**MIDWEST
MICRO Inc.**

MAIL ORDER: Add \$1.50 shipping and handling (\$3.50 for C.O.D.); VISA/Mastercard accepted (card# and exp. date). MO residents add 5.825% sales tax. Foreign orders payable U.S. Bank ONLY; add \$5 shipping.

311 WEST 72nd ST. • KANSAS CITY • MO • 64114

Macintosh is qualitatively distinct from any other personal computer. It has defined a new tier of the market.

This definition had happened *not* because of its 32-bit architecture, its 1 to 2 million instructions-per-second speed, or its price, but simply because of its functionality. For years the industry has been telling us that computers are easy to use. Macintosh finally came out to fulfill that promise.

But will Macintosh be successful? I hope so. Apple appears dedicated to supporting third-party software developers, and several powerful languages are available for users who like to create their own programs.

Back On The Right Track

There is another reason I hope Macintosh is successful. This country was built on the concept that people with good ideas could compete in the open marketplace. This spirit of open competition guaranteed not only that the customer got a good deal, but that technology would improve as newer and better products were developed. If, by pure force of corporate identity, we can be convinced to drop our high standards of cost-effective performance, we can kiss the free enterprise system goodbye.

Macintosh is more than a computer—it is a statement in response to the clearly stated needs of the consumer. How will *we* respond? **C**

INSIGHT: Atari

Bill Wilkinson

As I write this, I have just returned from the Las Vegas Comdex show.

Comdex stands for "COMputer Dealers' EXposition," but it is really a show for those who would sell *to* the computer dealers. And sell they did. Everything from magic acts to talking robots to sit-down demonstrations (very welcome after walking through literally acres and acres of booths). And, of course, IBM was there in force, occupying an entire building and demonstrating the usual stuff on the PC and, not surprisingly, some me-too-ish software on the PCjr.

Compatible Disk Drives

The only Atari-compatible hardware products that I saw at Comdex were some disk drives (though I understand that one or two graphics tablets were shown there, also). And that, of course, brings up my next topic.

When you consider the fact that Atari doesn't even make a double-density disk drive, it's more than a little surprising and pleasing to discover the amazing degree of compatibility exhibited by the various non-Atari disk drives.

Since OSS provides the disk operating system (DOS XL) which many of the drive manufacturers supply with (or as an option to) their disks, I can't make judgments as to quality, reliability, etc., without an obvious conflict of interest. I can, however, comment on the features common to all Atari-compatible drives (except those made by Atari itself).

The 815 Drive's Legacy

Historically, the reason for the compatibility is the ill-fated Atari 815 drive. For those of you relatively new to the world of Atari, that was the dual, double-density disk drive announced by Atari for delivery in early 1982. Notice the word "was."

Although never produced in quantity, the 815 survived long enough to cause Atari, Inc., to produce DOS 2.0d ("d" for double), and a few lucky people even have a copy of it. (I'm not lucky.) In fact, even Atari DOS 2.0s can access an 815 style double-density drive for most functions (just don't try to copy files or duplicate disks).

The folks at Percom Data Corporation, though, didn't know the 815 was going to die when they started designing their double-density drives. They did, however, want a way to switch from single to double density without having to physically flick a switch. Hence the configuration block was born. Give Percom credit.

Give the other manufacturers credit, also, for recognizing the Percom system as a viable and usable standard. Would you be surprised to find that the same double-density DOS XL diskette works unchanged in drives or controllers from (in alphabetical order) Amdek, Astra, Concorde, Indus, Micro Mainframe, NCT, Percom, Rana, SPI, and Trak? If you are *not* surprised, you are not aware of the hodgepodge of the CP/M world.

Each of the companies mentioned can tell you of the advantages of their drives or controllers.

A final comment on the configuration block

scheme mentioned above. A controller capable of implementing all the options of the configuration block can, in theory, support virtually any size disk drive. At Comdex I saw floppy disk drives with densities over a megabyte. Yum.

XL Compatibility

I have received more than a little correspondence from readers asking what they can do about the lack of software compatible with their 1200XL (and, now, the 600XL and 800XL). Up until now, my stock answer has been that they should go beat on the heads of the software manufacturers (the ones who didn't follow Atari's rules).

Now, though, there is a little relief in sight. Atari has, at long last, made available something known as the Atari Translator Disk. This disk, when booted from any 810-compatible drive into any XL machine with 64K of RAM, will (for all practical purposes) turn your XL computer into a non-XL Atari 800. Virtually all software, including protected games and the like, will then boot and run properly. (Of course, you don't turn the power off to boot anymore.)

For those who are stuck with incompatible software, this seems like a neat solution. For those who are stuck with incompatible software and no disk drive, this looks like a frustrating solution. Point of interest: I do believe that this software could be loaded via cartridge instead, since one need not turn off the power to change or remove cartridges on an XL machine. Atari, are you listening?

Anyway, if you need the disk, check with your local authorized Atari dealer. If he doesn't have it, hasn't heard of it, or is nonexistent, try Atari's customer service department.

Reading Binary Files

In March, I presented a short program in Atari BASIC which would read a binary object file directly into the memory locations it was originally assembled for (or saved from).

This month, I will start to parallel that listing in machine language. Please understand that this may not be the fastest or easiest way to perform the task. I use the BASIC parallel method as a way of making the program understandable to those who are just beginning to learn machine language.

As a first step, you might look through the listing, noting where the BASIC line equivalents are. They are easy to find. Starting at line 1000, any line number ending in 00 is a comment line which reflects the line in the BASIC program which I presented last month. Note, also, that the line numbers in this listing are 10 times the BASIC line numbers (simply for convenience and readability).

While examining the listing, you probably noted that there seem to be

code than otherwise. In truth, this simple pseudo-BASIC program does indeed require a fairly substantial amount of support. The support is in two forms: definitions of variables (including buffers) and I/O subroutines.

A Page 6 Assembly

You may also have noticed that I assembled the listing in the infamous page 6 memory block. I plead guilty. Actually, in testing this program, I assembled it twice: once at \$600, as shown, and once at \$6000 (just by changing line 110). I then used the \$600 version to read in the \$6000 version, and it worked!

Anyway, since I will be giving you complete source code here, I don't feel too guilty. Obviously, you can change line 110 to anything you wish if you need to stay out of page 6.

There are two other "cheats" in this listing. In line 220, I place NAME at location \$580; and, in lines 250 and 270, I place START and ADDR at location \$CE. Are these locations truly safe to use? In general, no. If you have been reading my series on self-relocatable code, you know that there are no truly safe locations. But for the purposes of this demonstration, I think we can use them as is, since they are compatible with usage by the Atari Assembler Editor (and MAC/65 and—I believe—AMAC) and Atari BASIC (and BASIC XL but not Microsoft BASIC).

One other comment before we begin analyzing the operation of the listed code. If you wish to use this program as a callable USR routine from Atari BASIC, you need to add this line:

125 PLA; clean up stack for BASIC

BASIC And ML Compared

Now, onward and downward, into the depths of machine language. I will discuss the lines which I feel are relevant and important by line number.

Line 130. We could have accomplished the same thing by giving a RUN address at the end of the listing, but this gets us started in a visible way.

Line 210. Note the use of the \$9B (an ATASCII RETURN code) to terminate the message. The 0 is for safety and because I am paranoid.

Double Usage

Lines 230, 240, 260. If you consider LOW and HIGH together, they form a 16-bit word. Since QUIT needs to be a word, why not join usage? This is not recommended procedure, but it works if you are careful.

Lines 250, 270. This isn't surprising if you think about the fact that line 310 in the BASIC code could have been written as FOR START=START TO QUIT, thus eliminating the need for the extra variable, ADDR.

Lines 300-321. These ...

OS listings and *Inside Atari DOS* though the actual mnemonics may differ slightly.

Lines 550-566. When you get to this routine, it expects the OS channel designator (which is 16 times the Atari BASIC file number) in the X register, the command value in the A register, and the address of the buffer to use in the Y register (low byte) and on the stack (high byte). The routine assumes that you will not be doing I/O which requires over 255 bytes of buffer (a valid assumption for this program, but not for all circumstances).

Checking For Errors

CMDJOIN sets up the appropriate IOCB and calls

CIO to do the real work. It returns the error status to the user in A, Y, and the flags. In this program, only OPEN looks for the error status. (Because PRINT and INPUT to/from channel zero had better work, and if CLOSE fails it's too late anyhow.)

Lines 500-545. These are the various I/O entry points. Note that they expect the X and Y registers set up as in CMDJOIN. They assume that the high byte of the buffer address is in A and push it on the stack to make room for the command byte. They are simple and effective.

Next month we'll look at the rest of this listing.

Load A Binary Object File

```

0100      .TITLE "Binary Object File Loader for COMPUTE!"
0101 ;
0102 ;
0103 ; a binary object file loader in assembly language
0104 ;
0000      0110      *= $0600      ; an arbitrary location
0600      0120 BEGIN
0600 4C6006      0130      JMP BEGINWORK ; skip data and subroutines
0140 ;
0150 ;::::::::::::::::::::::::::::::::::
0160 ;
0170 ; variables and buffers
0180 ;
0190 ; defined in order encountered in BASIC program
0200 ;
0603 57484154      0210 MESSAGE .BYTE "WHAT FILE TO LOAD ?", $9B, 0
0607 2046494C
060B 4520544F
060F 204C4F41
0613 44203F9B
0617 00
      =0580      0220 NAME = $0580      ; buffer for file name (see text)
0618 00      0230 LOW .BYTE 0      ; low byte of address
0619 00      0240 HIGH .BYTE 0      ; high byte of address
      =00CE      0250 START = $CE      ; although START could be anywhere,
      0251 ; ADDR (see below) needs zero page
      =0618      0260 QUIT = LOW      ; accomplishes line 270 of BASIC program
      =00CE      0270 ADDR = START      ; accomplishes part of FOR statement
      0271 ; in line 310 (see text)
0300 ;::::::::::::::::::::::::::::::::::::
0301 ;
0302 ; system equates, etc.
0303 ;
      =0340      0304 IOCB = $0340      ; where IOCB #0 is
      =0342      0305 ICCOM = $0342      ; the command byte
      =0344      0306 ICBADR = $0344      ; buffer addr
      =0348      0307 ICBLEN = $0348      ; buffer length
      =034A      0308 ICAUX1 = $034A      ; aux 1 byte (open mode)
0310 ;
      =0003      0311 CMDOPEN = 3      ; the open command
      =000C      0312 CMDCLOSE = 12      ; the close command
      =0009      0313 CMDPRINT = 9      ; put a text line
      =0005      0314 CMDINPUT = 5      ; get a text line
      =0007      0315 CMDGET = 7      ; get a binary byte or block
0320 ;
      =E456      0321 CIO = $E456      ; the master I/O routine for Atari OS
0498      .PAGE "      Major I/O Subroutines"
0499 ;
0500 ;::::::::::::::::::::::::::::::::::::

```

```

0501 ;
0502 ; the subroutines used by our program
0503 ;
0510 ; --- perform an OPEN function ---
0511 OPEN
0512     PHA                ; save high byte of address
0513     LDA #CMDOPEN
0514     BNE CMDJOIN
0515 ;
0520 ; --- perform a CLOSE function ---
0521 CLOSE
0522     PHA                ; save high byte of address
0523     LDA #CMDCLOSE
0524     BNE CMDJOIN
0525 ;
0530 ; --- perform a PRINT function ---
0531 PRINT
0532     PHA                ; save high byte of address
0533     LDA #CMDPRINT
0534     BNE CMDJOIN
0535 ;
0540 ; --- perform an INPUT function ---
0541 INPUT
0542     PHA                ; save high byte of address
0543     LDA #CMDINPUT
0545 ;
0550 ; code common to OPEN, CLOSE, PRINT, INPUT
0551 ;
0552 CMDJOIN
0553     STA ICCOM,X ; the command value
0554     PLA                ; recover high byte of addr
0555     STA ICBADR+1,X ; and set it up in iocb
0556     TYA
0557     STA ICBADR,X ; ditto with low byte of addr
0558     LDA #0
0559     STA ICBLN+1,X ; set up a maximum length
0560     LDA #255
0561     STA ICBLN,X ; of 255 bytes
0562     JSR CIO           ; then do the I/O operation
0563     TYA                ; any boo-boo's ?
0564     RTS                ; back to caller with error, if any
0565 ; (note that only OPEN call provides for
0566 ; an error...see text)
0598     .PAGE "          The GET Subroutine"
0599 ;
0600 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
0601 ; the GET routine...it's special
0602 ;
0603 GET
0604     LDA #CMDGET
0605     STA ICCOM,X ; set up for GET command
0606     LDA #0
0607     STA ICBLN,X ; by zeroing the length field,
0608     STA ICBLN+1,X ; ...we get a single byte to A
0609     JSR CIO           ; let OS do the work
0610     INY
0611     DEY
0612     BMI BADGET ; check status "invisibly"
0613     RTS                ; oops
0614 ; (remove BMI for caller to get status instead)
0615 ;
0616 BADGET
0617     PLA
0618     PLA                ; this is a cheat
0619     JMP LINE400 ; but it works
0689     .PAGE "

```

INSIGHT: Atari

Bill Wilkinson

The assembler listing which accompanies this article is a set of patches to Atari DOS 2.0s. If you own an Atari 1050 drive, these patches will allow you to use it in "enhanced density" mode.

Before we get started with the listing and its explanation, though, let's look at a new tidbit.

Bye-Bye BASIC

Are you an 800XL owner? Do you have an unprotected diskette which boots a machine language program via an AUTORUN.SYS file? Would you like to avoid pushing the OPTION button? Are you willing to follow a few simple steps to do so?

Your 800XL enables and disables the built-in BASIC by changing the contents of location \$D301 (54017). In Atari 400s and 800s, this location is usually used to input the state of joysticks 3 and 4. In an 800XL, this port controls various system hardware configurations.

For example, bit 0 of \$D301 controls whether the OS ROM is active or whether you are using the RAM underneath it. And—guess what—bit 1 of \$D301 controls whether the built-in BASIC is active or not. Specifically, the following table applies:

Bit 0	= 1	OS ROM enabled
	0	OS ROM disabled, RAM enabled
Bit 1	= 1	Atari BASIC disabled, RAM enabled
	0	Atari BASIC enabled

At least one of the other bits in \$D301 is used (to control whether or not the diagnostic ROM is enabled), but the "normal" values for \$D301 are either \$FF (BASIC disabled) or \$FD (BASIC enabled).

No Option Button

So all we need to do is add a couple of instructions to our AUTORUN.SYS file, to select RAM instead of BASIC, and we will no longer have to hold down the OPTION button. For example, we might add:

```
LDA #$FF
STA $D301
```

And, yet, there is an easier way. Remember, Atari LOAD files may consist of multiple segments, each of which starts with a start address and an end address. The entire file starts with a pair of \$FF bytes, but it doesn't hurt if there are

extra \$FF header bytes in front of other segments.

So consider: If we specify that we have a LOAD file which starts at location \$D301 and ends at location \$D301, the DOS file loader will try to load (and thereby store) a single byte at location \$D301. This is equivalent to storing a byte via our program.

Disabling BASIC

So simply use the following steps to modify your AUTORUN.SYS to disable the built-in BASIC:

Under Atari DOS 2.0s:

1. Boot your DOS disk while holding down the OPTION button.
2. Put the disk containing the AUTORUN.SYS you want to modify into drive 1.
3. Use the E option from the DOS menu. When prompted for old and new filenames, respond:
D:AUTORUN.SYS,AUTORUN.OLD
4. Use the K option from the DOS menu. When prompted for filename, starting address, etc., respond:
D:AUTORUN.SYS,D301,D301
5. Use the C option from the DOS menu. When prompted for from and to filenames, respond:
D:AUTORUN.OLD,AUTORUN.SYS/A

Under OS/A+ or DOS XL:

1. Boot your DOS disk while holding down the OPTION button. If the DOS XL menu appears, use the Q option.
2. Put the disk containing the AUTORUN.SYS you want to modify into drive 1.
3. Type the command:
RENAME AUTORUN.SYS AUTORUN.OLD
4. Type the command:
SAVE AUTORUN.SYS D301 D301
5. Type the command:
COPY -AF AUTORUN.OLD AUTORUN.SYS

And that's it. Your AUTORUN.SYS file should now be ready to use.

Check The Pointers

Caution! Even though the built-in BASIC is now disabled, HIMEM (the contents of location \$2E5) and RAMTOP (contents of location \$6A) will still reflect the 40K byte configuration where BASIC is present. If your program pays attention to one or both of these two values, it would also be worth performing the following steps:

1. Change RAMTOP to reflect the full 48K bytes.
2. Close channel zero (the screen editor).
3. Open channel zero for the E: device.

These steps will insure that all 48K bytes of accessible RAM are in use by your program. I won't go into how to accomplish these here and now. Write if you would like me to show how to code those steps in machine language.

Coming Attractions

A project related to this, which I hope to implement in an upcoming column, would be an "M:" device driver. Once upon a lifetime ago, in this column, I presented such a driver. It used the "excess" memory (between the top of a BASIC program and the bottom of the graphics screen) as a pseudodevice.

I would like to do the same thing again, but this time use the extra memory under the OS ROMs or under the built-in BASIC as a superfast RAM disk. Stay tuned for further developments.

DOS 2.0s For Enhanced Density 1050s

First, I would like to point out that the task of reconfiguring Atari DOS 2.0s for an enhanced density 1050 is difficult. I would also like to note that it is *extremely* difficult (if not impossible) to finish the task if you have only one drive.

So, may I suggest that you cooperate with a friend and his drive if you have only one of your own. If your friend's drive is an 810 or a non-Atari drive, it should be set up as drive 1. Your 1050 should be set up as drive 2.

Also, you should use an assembler capable of placing its object code directly in memory. (For example, AMAC—the Atari Macro Assembler—cannot be used for this job.) This is because loading the DOS-modifier code from a disk will use DOS itself, and you are almost guaranteed to run into conflicts. Atari's Assembler Editor cartridge, the old OSS EASMD, OSS's MAC/65, and (I believe) SYNASSEMBLER will all work properly (though the syntax for SYNASSEMBLER may vary a bit from what I show here).

You should boot a normal Atari DOS 2.0s disk, making sure that you can access a normal single diskette in drive 2 (at least to the point of making sure you can list its directory). Be sure

you have at least two (2) blank or junk disks ready and at hand. Then begin.

Patching DOS

Type in the program, as shown herein. You may use automatic line numbering if you wish. Type in just the part from the right of the line numbers. LIST or SAVE the source code to disk and then assemble it. Check it against the listing given here. Do not proceed until you are reasonably sure that you have typed it in and assembled it correctly.

Then change line 1000 to read:

```
1000 .OPT NOLIST,OBJ
```

and assemble the code once more.

Voilà! DOS has been patched!

But, because DOS's DRVTLB has changed format, you *must* now hit the SYSTEM RESET key. Then give the DOS command from your assembler. Assuming that you get to the DOS menu (and if you don't, you did something wrong), it would probably be a good idea to immediately format (menu option I) a blank disk in drive 1 and write the DOS files (option H).

Implementing Enhanced Density

Now comes the tricky part. The way we have patched DOS 2.0s, DOS automatically checks each drive at power-on (or SYSTEM RESET) time to find its current configuration (single density, double density, or enhanced density). But the 1050 assumes it is in single-density mode unless you have inserted an enhanced-density diskette. So, up until now, DOS thinks it is working with all single-density disk drives. How do we change its mind?

The easy way: Turn your power off, put your BASIC (or BASIC XL) cartridge into your machine, and turn the power back on, thus booting the disk we just formatted and wrote DOS files to. Insert a blank disk into the second drive (your 1050). From BASIC, give the following command:

```
XIO 254,#1,0,34,"D2:"
```

If you are a faithful reader, you will recognize that as the format command, given from BASIC. But the 34 in the next-to-last position is new! That's right. As we have patched DOS, a nonzero value given in AUX2 is assumed to be the format command value to be sent to the disk drive. The *only legal values* here are 33 (for single density, à la 810 drives) and 34 (for 1050 enhanced density)!

Now drive 2 contains what we hope is an enhanced-density diskette. Once more, hit SYSTEM RESET so that DOS will recognize the new density. Then give the DOS command from BASIC. Once in DOS, use the H menu option to write the DOS files to drive 2.

If you have performed all these steps correctly, you should now have a bootable enhanced-density diskette in drive 2. You might want to change your 1050 back to being drive 1 and try to boot from it with this new diskette.

Simpler Commands

The beauty of this system is that, once you have created this one enhanced-density master, you may make new enhanced-density masters by using just the I and H commands from the DOS menu.

There is, however, one potential problem. How do you copy files from an old single-density disk to a new enhanced-density disk? For now, the only practical way is to borrow a second drive and have one of each type of disk on your system. There may be ways around even this problem. We'll see.

Patching Other DOS Versions

The patch program given here will also work on all versions of OS/A+ and DOS XL from 1.2 to 2.3 (except that it will *not* patch the DOSXL.SYS versions).

The procedures are almost the same, but it is significantly easier to use a single drive. Try the following if you have only a single disk, on which you boot OS/A+ or DOS XL:

1. Type in, save, and check out the patch listing as described above.
2. Hit SYSTEM RESET. If you end up back in an assembler cartridge, type a DOS command.
3. From the D1: prompt, use an INIT command. Or use the I option from the DOS XL menu.
4. Use Option 1 (on a blank disk) or 3 (on an existing disk) of INIT. Use Option 4 to return to DOS.
5. Insert a BASIC cartridge. Reboot from the disk you just INITED.
6. Type the following BASIC command:
XIO 254,#1,0,34,"D1:"
7. Hit SYSTEM RESET after the formatting is finished. If you are not then in the BASIC cartridge, use the CAR command.
8. Type the following BASIC command line:
OPEN #1,8,0,"D1:DOS.SYS" : CLOSE #1

The reason the procedure works on a single drive is that neither OS/A+ nor DOS XL requires the DUP.SYS file of Atari DOS. The disk initialization can thus be performed entirely from BASIC.

Patches To Atari DOS 2.0s

```

1000      .OPT LIST,NO OBJ
1010      ;::::::::::::::::::::::::::::::::::::::::::::::::::
1030      ; PATCHES TO ATARI DOS 2.0S
1040      ;
1050      ; THESE PATCHES ALLOW AN ATARI 1050 DRIVE
1060      ;   TO UTILIZE ENHANCED DENSITY UNDER
1070      ;   DOS 2.0S, TO A MAXIMUM OF 965 FREE SECTORS
1100      ;::::::::::::::::::::::::::::::::::::::::::::::::::
1110      ;
1120      ; EQUATES -- TAKEN FROM THE LISTING OF
1130      ;               ATARI DOS AS PUBLISHED IN
1140      ;               "INSIDE ATARI DOS"
1150      ;               FROM COMPUTE! BOOKS
1160      ;
=1311      1170 DRV TBL = $1311
=1301      1180 CURFCB = $1301
=0048      1190 ZSBA = $48
=11DB      1200 DERR1 = $11DB
=12FE      1210 DRV TYP = $12FE
=0021      1220 DCBCFD = 'I
=0302      1230 DCBCMD = $0302
=0045      1240 ZDRVA = $45
=0A4A      1250 NOBURST = $0A4A
=0A4C      1260 WRBUR = $0A4C
=0D18      1270 XFORMAT = $0D18
=0BD6      1280 XFV = $0BD6
=1372      1290 Z = $1372
=034B      1300 ICAUX2 = $034B
=1382      1310 FCBOTC = $1382
1340      ;::::::::::::::::::::::::::::::::::::::::::::::::::

```

The rest of this listing will appear in "Insight: Atari" next month.

INSIGHT: Atari

Bill Wilkinson

Last month we discussed how to make programs designed for the Atari 400 and 800 load and run automatically on the new XL series without having to hold the option key down. We also looked at a way to make patches into Atari DOS 2.0s to enable it to work with the new enhanced density 1050 disk drive. The procedure is easy, but requires two disk drives. Just type in the source code (the portion printed last month and the continuation found in this issue) using an assembler capable of placing its object code directly in memory. Assemble it after LISTing or SAVEing the source code to disk. After assembling it once, change line number 1000 to read:

```
1000 .OPT NOLIST,OBJ
```

and assemble the code once more.

DOS should now be patched. Hit the SYSTEM RESET key and give the DOS command from your assembler. You should now be in the DOS menu (if you're not, something has gone wrong). Format a new disk using option I and then write the DOS files using option H. This will insure that everything is right and will give

you a safe copy of your newly patched DOS.

The Tricky Part

There's one more step necessary to finish the procedure. Turn off your computer, put your BASIC (or BASIC XL) cartridge into your machine, and turn the power back on, thus booting the disk that was just formatted. Place a blank diskette into the 1050 drive that you are using as drive 2 and, from BASIC, type the following command:

```
XIO 254,#1,0,34,"D2:"
```

Drive 2 should now contain an enhanced-density diskette. Now hit the SYSTEM RESET key so that DOS will recognize the new density. Finally, go into DOS and write the DOS files to the new diskette (D2), using option H from the menu.

If everything has been done properly, drive 2 should now have an enhanced-density diskette containing the patched DOS. Once you have this master completed, creating others is simple and can be done with the I and H options in the DOS menu.

Patches To Atari DOS 2.0s

```

1350 ;
1360 ; BEGIN THE ACTUAL PATCHES
1370 ;
1380 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
1390 ;
1400 ; This patch allows either $21 or $22 as
1410 ;   a format command.
1420 ;
1430   *= $07B5
1440 PATCHFORMAT
1450   LDA #$20      ; format cmds are $21 or $22
1460   AND DCBCMD    ; is this a format cmd?
1470   BNE $07BE     ; bit $20 is set, so yes...read
1480 ;
1490 ; This patch modifies the drive type
1500 ;   reported by DINIT for use in DRVTBL
1510 ;
1520   *= $0819
1530 PATCHINIT
1540   LDA $02EA     ; get drive status
1550   ASL A          ; and this sequence...
1560   PHP           ; ...will serve to
1570   ASL A          ; ...convert the status
1580   ROL A          ; ...$00, $20, and $80
1590   ROL A          ; ...to the more usable
1600   ROL A          ; ...$00, $01, and $80
1610   PLP           ; (more usable because what we
1620   ROR A          ; want are $01, $02, and $81)

```



```

1630 ;
1640 ;
1650 ; This patch allows SETUP to call the main
1660 ; patch-it-all-up routine
1670 ;
0824 1680      *= $1184
1690 ; patch to SETUP code
1700 ;
1184 BE1113 1710      LDX DRVTBL,Y
1187 200115 1720      JSR PATCHSETUP ; the real work
118A A8      1730      TAY
118B F04E    1740      BEQ DERR1
1750 ;
1760 ;::::::::::::::::::::::::::::::::::::::::::::
1770 ;
1780 ; The major patch:
1790 ; Here we determine the type of drive for
1800 ; the current operation and patch various
1810 ; locations (including LDA # instructions)
1820 ;
118D 1830      *= $1501
1501 1840 PATCHSETUP
1501 8E7213 1850      STX TRUETYPE ; save true drive type
1504 E8      1860      INX ; convert 0 or 1 to 1 or 2
1505 8A      1870      TXA
1506 2903    1880      AND #$03 ; mask off 1050 bit
1508 8DFE12 1890      STA DRVTYP ; ...and save it
150B 48      1900      PHA ; and keep it for later return
1910 ; now we set up the tricky stuff
1920 ;
1930 ; we need different VTOC bases and sizes
1940 ; and different disk sizes
1950 ;
150C A00A    1960      LDY #$0A ; 810: start of vtoc
150E A964    1970      LDA #90+$0A ; 810: end of vtoc bytes
1510 A221    1980      LDX #DCBCFD ; 810: format command
1512 2C7213 1990      BIT TRUETYPE ; test drive type
1515 1005    2000      BPL SGLDBLJOIN ; 810, all ok
1517 A006    2010      LDY #$06 ; 1050: start of vtoc
1519 A980    2020      LDA #122+$06 ; 1050: end of vtoc bytes
151B E8      2030      INX ; 1050: format command is ""
2040 ;
2050 ; now store these values into code (shudder!)
2060 ;
151C 2070 SGLDBLJOIN
151C 8E230D 2080 ;
2090      STX $0D23 ; where format command is loaded
2100 ;
2110 ; the various uses of START-OF-VTOC
2120 ;
2130 ;
151F 8C800D 2140      STY $0D80 ; in deallocation of boot
1522 8CEE10 2150      STY $10EE ; in FRESECT
1525 8C4211 2160      STY $1142 ; in GETSECTOR, displacement
1528 88      2170      DEY
1529 8C0711 2180      STY $1107 ; at start of GETSECTOR
2190 ;
2200 ; and the uses of END-OF-VTOC
2210 ;
152C 8D0A11 2220      STA $110A ; check end in GETSECTOR
152F 8D7A0D 2230      STA $0D7A ; a CPY in format code
1532 98      2240      TYA

```

1533 18	2250 CLC	
1534 692E	2260 ADC #46	; adjust for ...
1536 8D840D	2270 STA \$0D84	; the directory dealloc in fmt
1539 AE0113	2280 LDX CURFCB	; recover original value
153C 68	2290 PLA	
153D 60	2300 RTS	
	2310 ;	
	2320 ;	::
	2330 ;	
	2340 ;	
	2350 ;	This is another major patch...
	2360 ;	instead of using set values for VTOC
	2370 ;	info, we pick from one of two tables
	2380 ;	
	2390 ;	
153E	2400	*= \$0D37
0D37	2410	PATCHXFORMAT
0D37 1027	2420	BPL XF0 ; same source, but XF0 has moved
	2430 ;	
	2440 ;	
0D39	2450	*= \$0D52
0D52	2460	TBL810
0D52 02	2470	.BYTE 2
0D53 C302	2480	.WORD 707,707
0D55 C302		
0D57 00FF	2490	.BYTE 0,\$FF
0D59	2500	TBL1050
0D59 02	2510	.BYTE 2
0D5A C503	2520	.WORD 965,965
0D5C C503		
0D5E 00FF	2530	.BYTE 0,\$FF
	2540 ;	
	2550 ;	
	2560 ;	we have moved the label xf0
	2570 ;	...to make room for the tables
	2580 ;	
0D60	2590	XF0
0D60 A000	2600	LDY #0
0D62 B9520D	2610	XF01 LDA TBL810,Y
0D65 2C7213	2620	BIT TRUETYPE
0D68 1003	2630	BPL TYPEOK
0D6A B9590D	2640	LDA TBL1050,Y
0D6D	2650	TYPEOK
0D6D 9145	2660	STA (ZDRVA),Y
0D6F C8	2670	INY
0D70 C007	2680	CPY #7
0D72 D0EE	2690	BNE XF01
	2700 ;	
0D74	2710	XF02
0D74 9145	2720	STA (ZDRVA),Y
0D76 C8	2730	INY
0D77 10FB	2740	BPL XF02
0D79 EA	2750	NOP
0D7A EA	2760	NOP
0D7B EA	2770	NOP
0D7C EA	2780	NOP
	2790 ;	
	2800 ;	This patch allows the user to choose
	2810 ;	diskette type for formatting via
	2820 ;	the 'XIO 254' command
	2830 ;	
0D7D	2840	*= XFV ; Where the format vector is

```

5 4C7313      2850      JMP XFVPATCH
                2860 ;
                2870 ; The label 'Z' designates some unused
                2880 ; memory in the original DOS 2.0s
                2890 ;
                2900      *= Z
                2910 TRUETYPE .BYTE 0 ; Where PATCHSETUP saves true disk type
                2920 ;
                2930 ; This code becomes the beginning of
                2940 ; the FORMAT code
                2950 ;
                2960 XFVPATCH
                2970      LDA ICAUX2,X ; Get AUX2 value
                2980      BEQ XFVP2 ; zero...don't do anything
                2990      STA $0D23 ; non-zero...assume it is type of format
                3000 XFVP2
                3010      JMP XFORMAT
                3020 ;
                3030 ;
                3040 ;::::::::::::::::::::::::::::::::::::::::::
                3050 ;
                3060 ; end of patches for 1050 drive
                3070 ;
                3080 ;
                3090 ; BEGIN patches for BURST I/O
                3100 ;
                3110 ; from COMPUTE!, July, 1982
                3120 ;
                3130 ;::::::::::::::::::::::::::::::::::::::::::
                3140 ;
                3150      *= $0A1F
                3160 ;
                3170 ; first, patch the code where WTBUR used to be
                3180 ;
                3190 WTBUR
                3200 BURSTIO
                3210      LDA FCBOTC,X ; open type code
                3220      EOR #$0C ; check for mode 12 (update)
                3230      BEQ NOBURST
                3240      ROR A ; move carry to MSB of A-reg
                3250      NOP ; filler only
                3260 TBURST
                3270 ;
                3280 ; ... and STA BURTYP remains...but
                3290 ; BURTYP is negative if BURSTIO was
                3300 ; called from GET-BYTE and positive
                3310 ; if it was called from PUT-BYTE
                3320 ;
                3330      *= $0A41
                3340 ; so we must patch here to count for the sense
                3350 ; of BURTYP being inverted from original
                3360 ;
                3370      BPL WRBUR
                3380 ;
                3390      *= $0AD4
                3400 ;
                3410 ; finally, we must patch the GET-BYTE call
                3420 ; so that it JSR's to new location
                3430 ;
                3440      JSR BURSTIO
                3450 ;
                3460      .END

```


INSIGHT: Atari

Bill Wilkinson

Let me start my fourth year by discussing the biggest event in Atari history since the introduction of the 800 Home Computer. But first a moment's reflection on my years with COMPUTE!.

Sometime about three years ago, I was reading COMPUTE!'s "The Readers' Feedback" column when I noticed a couple of questions about Atari computers which the editorial staff hadn't answered. And I also noticed a question which a reader had answered incorrectly.

I reacted. I phoned COMPUTE! and, for reasons best known to himself alone, a nice gentleman by the name of Richard Mansfield listened to my ranting and raving. I started to write for COMPUTE!.

Since then, I have written many columns and have covered a wide range of topics. But now I feel that it's time to change the style of this column. When I started, I intended to answer two or three questions a month and perhaps add a tidbit of my own. Lately, though, I've paid less attention to what you, my readers, want and have shown you some exotic but (perhaps for many of you) uninteresting programs, etc.

I am going to try to revive the chitchat style of this column. It will be more fun for me to write like that again and, I hope, more fun for you to read.

Whither Atari?

As I write this, only a few days have passed since the bombshell exploded: Jack Tramiel bought Atari! I don't see how I could avoid commenting on this—even if I wanted to.

By the time you read this, some of the things I will speculate on here will have been reduced to the role of mere facts or—equally likely—humorous fiction. Nevertheless, I would like to try to play the crystal ball game. Bear with me, please, as I make some predictions:

Nobody pays a quarter of a *billion* dollars (even 1991 dollars) for a name alone. If Mr. Tramiel doesn't produce and sell some (many?) of Atari's current and/or soon-to-be-current products, he will have bought nothing at all (since the massive layoffs make it obvious that he has little use for the expertise of the people who *were* Atari).

What products will survive? Probably the 800XL. It's a good machine and can probably be cost-reduced to be truly competitive with the Commodore 64. It could well have an effective price/performance ratio for well into the next two years.

I'm not so sure about the peripherals. The disk drive, or a version of it, certainly. Printers, of course. The cassette recorder? It's a piece of junk, and everyone knows it. The much ballyhooed add-on box, with MSDOS, 80-column screen, 128K bytes, and an ice cream freezer? Maybe. But don't be surprised to see it licensed to a third-party, low-volume manufacturer. It's too difficult for a lean and mean company to support such a complex product.

A Fabulous Game Machine

What about the game machine side of Atari? Some have suggested that Mr. Tramiel will drop it like a hot potato. Baloney, I say. Why did he buy it then? Was he fooled by Warner and the ex-Atari management? I have heard Jack Tramiel called many things, but "stupid" is not one of them.

I have seen *and* played with the 7800 "Pro-System." It is a truly fabulous game machine (and it's even a fair computer, with 95 percent plus compatible Atari BASIC). Making it 100 percent compatible with the 2600 was a stroke of genius. When I buy one (and I will), I can keep

my collection of 20-odd 2600 cartridges. Though I suspect—having seen *Xevious* and *BallBlazer* and *Rescue on Fractulus* and *Robotron* and . . .—that they will be little used. Tens of millions, like the 2600? Maybe not. A few million? Definitely yes.

Now what about the supposedly all-important, superadvanced, already developed computer that Tramiel and associates are bringing to the game? Well, first of all, I don't know how far along that machine is. Designed? Almost certainly. Prototypes available? A good probability. Debugged and with software ready? Possible, but I seriously doubt it.

Personally, I expect to see an early prototype shown in January 1985, with "selling" models shown in June, both probably at the Consumer Electronics Shows.

An Atari Mac?

And what will this miracle machine, the savior of Atari as a "name" in the industry, look like? Ah, now there you've got me. I am skeptical about reports that it will be a "business" machine: Why buy a game company's name for such a scheme? But an integrated "noncomputer" such as the Macintosh? Sure! Maybe even "a computer for the rest of us" (Apple's Macintosh slogan) that is affordable by the rest of us.

Well, how did I score? Or is it still too soon to tell? I am more than a little interested in knowing the outcome.

Answered Letters

Several people, led by Lloyd Keller of Palmetto, Florida, wrote me about something I tossed in, offhand, in my June column. While discussing the Atari *Translator Disk*, I had said, "Of course, you don't turn the power off to boot anymore."

And why not? Because, on an 800XL or 1200XL, the *Translator Disk* software loads into the RAM which is shared with OS (the hidden, bank-selected RAM). It then switches out the ROM completely, leaving you with an Operating System in RAM which is much, much more compatible with the old Atari 800's OS. Many programs which will not work in XL machines suddenly work just fine.

However, since your OS is now in RAM, you certainly can't turn off the power in order to boot another disk (for example, a protected game). Similarly, some cartridges insist on the old OS before they will run. You can't turn off the power to plug them in and still retain the OS in RAM.

Thus, before running, the *Translator Disk* software allows you to change cartridges or cassettes and then tell it you are ready to do a coldboot. That's all there is to it.

Mr. Keller, however, pointed out that his manuals tell him not once but many times to never change a cartridge with the power on. Well, sometimes manuals tend toward the cautious side.

Point 1: Nobody sticks a cartridge out in plain sight and then designs the electronics so that a three-year-old's sticky fingers can zap the whole machine by removing it. Point 2: The OS in the XL machines has a complex cartridge-presence checker built in. It checks to see if a cartridge has been inserted or removed every time the OS is called or every 1/60 second.

The action of this checker varies between the 600XL/800XL and the 1200XL. On the former, it causes the machine to "hang" until you hit reset, at which point it does a power-on sequence. The 1200XL simply keeps trying to do a power-on sequence, over and over again, and could lock up as a result.

So my point remains: Someone could and should produce an inexpensive cartridge which would act like the *Translator Disk*, thus giving cassette-only owners access to a wider range of software.

The Loop That Shouldn't Work

Shame on all you loyal Atari readers. It took a couple of Commodore 64 owners to bring one of my mistakes to light. A. J. Bryant of Winnipeg, Canada, and David MacKenzie of Bethesda, Maryland, tried the FOR-NEXT nesting test that I presented in my March 1984 column on their 64 machines.

Lo and behold, the program works (it is supposed to fail). And Mr. MacKenzie even asked me if Microsoft knew something we didn't. Well. I couldn't take a challenge like that lying down, so I powered up our 64 (yes, we really do have one) and tried it myself. Hmm.

Then I tried it on my trusty 800XL. It worked there also! My face is red. Between the time I developed the test and the time I submitted it for publication by COMPUTE!, I tried to pretty it up. There is a variation on Murphy's law which is appropriate here: "If it ain't broke, don't fix it."

So Program 1 is the original FOR-NEXT test. It fails on all Atari computers. It fails on Commodore 64s and Applesoft. The normal mode of failure is to issue a NEXT WITHOUT FOR error at line 280.

At first, I was surprised when Apple Integer BASIC passed this test. But I soon discovered why: Integer BASIC doesn't treat nested FOR loops properly at all. Program 2 is another, simpler test I devised to smoke out BASICs which have this kind of problem, so let's take a quick look at it.

Line 10 and 20 simply set up a pair of nested loops. But then line 30 starts an outer loop over again (or at least an intelligent BASIC interpreter will think so, since we are reusing I as a loop variable). Thus, line 50 should cause an error, because starting the outer loop over should erase the information about the inner (FOR J) loop. Indeed, on all the BASICs I mentioned except Apple Integer BASIC, it does. With Integer BASIC, though, the error does not occur until line 60. Tch-tch.

If there are any BASICs which pass both these tests, I would like to hear of them. Thanks.

More Letters Next Month

I've already started wading through a pile of letters; and, although I obviously can't promise a response to every one, maybe I'll try to answer your question or comment next month. See you then.

Program 1: Original FOR-NEXT Test

```
100 REM IT IS NORMAL FOR THIS PRO
    GRAM TO STOP
101 REM WITH AN ERROR ON LINE 280
110 PRINT "I","J","I&J"
```

```
120 FOR I=1 TO 9
130   FOR J=1 TO 9
140     PROD=I&J
150     IF PROD > 14 THEN 200
160     IF PROD > 10 THEN 190
170     PRINT I,J,PROD
180   NEXT J
190 NEXT I
200 PRINT "J","I","J&I"
210 FOR J=1 TO 9
220   FOR I=1 TO 9
230     PROD = J&I
240     IF PROD > 14 THEN 300
250     IF PROD > 10 THEN 280
260     PRINT J,I,PROD
270   NEXT I
280 NEXT J
290 STOP
300 STOP
```

Program 2: FOR-NEXT Test 2

```
10 FOR I=1 TO 3
20   FOR J=1 TO 3
30   FOR I=10 TO 12
40     PRINT I,J
50   NEXT J
60 NEXT I
```

Lowest
Prices

MM
COMPUTER CENTER

Fast
Shipments

Indus GT-Atari	\$318	Software	
Atari 410 CASSETTE RECORDER	\$ 36	MPP1000C Modern Direct Connect	\$119 ⁰⁰
Dynax Letter Quality Printer by Brother	\$379	Commodore Automodem	\$88 ⁰⁰
Mannesman Tally Spirit 80	\$259	Maxell MD1 Box 10	\$19 ⁵⁰
APE Interface-Atari	\$ 58	Walbash Single Density Box 10	\$13 ⁹⁵
Commodore 1526 Printer	\$278	Visi-Calc Atari	\$59 ⁰⁰
Postage-Handling < Hardware \$10 Foreign \$14 Software 4.00 >			
VISA Mastercard		Call 1-800-543-5500 ASK FOR OPERATOR 525	
		COD Money Order	

Copy Atari 400/800/XL Series Cartridges to Disk and run them from a Menu

ATARI CARTRIDGE-TO-DISK COPY SYSTEM \$69.95

Supercart lets you copy ANY cartridge for the Atari 400/800/XL Series to diskette, and thereafter run it from your disk drive. Enjoy the convenience of selecting your favorite games from a "menu screen" rather than swapping cartridges in and out of your computer. Each cartridge copied by Supercart functions exactly like the original. Supercart includes:

- DISKETTE with:
 - COPY PROGRAM - Copies the cartridge to a diskette (up to 9 cartridges will fit on one disk.)
 - MENU PROGRAM - Automatically runs and displays a menu prompting user for a ONE keystroke selection of any cartridge on the disk.
- CARTRIDGE:
 - "Tricks" the computer into thinking that the original "copy protected" cartridge has been inserted.

To date there have been no problems duplicating and running all of the protected cartridges that we know of. However, FRONTRUNNER cannot guarantee the operation of all future cartridges. Supercart is user-friendly and simple to use and requires no modifications of your hardware. **PIRATES TAKE NOTE:** SUPERCART is not intended for illegal copying and/or distribution of copyrighted software. . . Sorry!!!

SYSTEM REQUIREMENTS:

Atari 400/800 or XL Series Computer / 48K Memory / One Disk Drive

Available at your computer store or direct from FRONTRUNNER. DEALER INQUIRIES ENCOURAGED

TOLL FREE ORDER LINE: (24 Hrs.) 1-800-548-4790 / In Nevada or for questions Call: (702) 786-4600

Personal checks allow 2-3 weeks to clear. M/C and VISA accepted.

Include \$3.50 (\$7.50 Foreign orders) for shipping.

FRONTRUNNER COMPUTER INDUSTRIES

316 California Ave., Suite #712, Reno, Nevada 89508 - (702) 786-4600

Others Make Claims. . . SUPERCART makes copies!!!

ATARI is a trademark of Warner Communications, Inc.

Program Your Own EPROMS

\$99.50

PLUGS INTO USER PORT.
NOTHING ELSE NEEDED.
EASY TO USE. VERSATILE.

promenade™

- Read or Program. One byte or 32K bytes!
- OR Use like a disk drive. LOAD, SAVE, GET, INPUT, PRINT, CMD, OPEN, CLOSE—**EPROM FILES!**
- Our software lets you use familiar BASIC commands to create, modify, scratch files on readily available EPROM chips. Adds a new dimension to your computing capability. Works with most ML Monitors too.
- Make Auto-Start Cartridges of your programs.
- The *promenade*™ C1 gives you 4 programming voltages, 2 EPROM supply voltages, 3 intelligent programming algorithms, 15 bit chip addressing, 3 LED's and NO switches. Your computer controls everything from software!
- Textool socket. Anti-static aluminum housing.
- EPROMS, cartridge PC boards, etc. at extra charge.
- Some EPROM types you can use with the *promenade*™

2758	2532	462732P	27128	5133	X2816A*
2516	2732	2564	27256	5143	52813*
2716	27C32	2764	68764	2815*	48016P*
27C16	2732A	27C64	68766	2816*	

► *Commodore Business Machines *Denotes electrically erasable types

Call Toll Free: 800-421-7731
In California: 800-421-7748

JASON-RANHEIM
580 Parrott St., San Jose, CA 95112

INSIGHT: Atari

Bill Wilkinson

Comparing Printers

After disk drives, probably the most frequently purchased peripheral for personal computer systems is a printer. But buying a printer is a lot harder than buying a disk drive. Usually your choice of drives is limited to the computer manufacturer's own unit plus a few produced by third-party companies. And despite some slight differences, they all deliver similar performance.

But printers are another story. There are hundreds of printers on the market for personal computers. Most of them can be made to work with your Atari. And they vary widely in terms of price, performance, features, and compatibility.

One of the main differences between printers is their printing speed. Usually this is measured in characters per second, abbreviated cps. By comparing the speed ratings, you can decide whether a certain printer is fast enough for your applications. But recently I discovered how misleading those speed ratings can sometimes be. It all started when those of us at Optimized Systems Software (OSS) began looking around for a new printer.

To begin, let me tell you that we have a rather unique requirement for a printer: We needed a good, fast, reliable printer which we could hook up to any of several computers. And, of course, it had to be compatible with all our software: several languages, four different operating systems, and a couple of word processors.

It is also time for a bit of history. For the last couple of years, our mainstay printer has been a venerable DEC LA-120 Decwriter. This is actually a printing terminal (remember, from the days of mainframe timesharing?) which operates via a serial RS-232-C connection at 120 cps. As reliable as this beast has proven to be, it has a few problems: Its print quality is marginal at best, without even descenders on lowercase letters; because it uses a serial instead of the more standard parallel interface, much software simply will not work with it; although it is rated at 120 cps, it is actually capable of only about 105 to 110 cps when printing typical documents.

At the time, the only other printers we had (or had significant experience with) were a Diablo daisywheel (also serial, at 30 cps), an Atari 825 (rated at 60 cps), and a C. Itoh Prowriter

(rated at 120 cps). All had performed adequately (or, in the case of the Prowriter, more than adequately), but all were too slow for our purposes.

And, of course, software compatibility was another big issue. Our primary problem in the past had been that some of our computers transmitted a linefeed after a carriage return (for example, the CP/M based machines), while others (our Atari computers) did not. We were well aware, also, that more problems would be coming as we acquired more software and wanted more capabilities.

Instantaneous Vs. Continuous Speed

For the sake of compatibility then, the first printer that came to mind was the Epson MX-80. Why? Simply because it is used on so many machines with so much software. Yet we immediately rejected the MX-80. Rated at only 80 cps, it is simply way too slow for our applications.

So we started looking for a *fast* printer which would be largely compatible with the MX-80. To make a long story short, we bought an Epson FX-100, a wide-carriage version of the FX-80. Imagine our surprise when this printer, rated at 160 cps, was only marginally faster than the Prowriter and actually *slower* than the Decwriter!

It turns out that with few exceptions, the printer speeds published by manufacturers and often faithfully reported by magazines are the *maximum instantaneous* speeds of which a machine is capable. This instantaneous speed rarely correlates to the actual number of lines a printer will produce in a minute.

What's more, even those companies which do admit that speed ratings are maximum values employ other claims to suggest that their printer is faster than the competition. For example, many claim that because their printers are *bidirectional* or *logic-seeking*, they are faster than the old-fashioned machines which print in only one direction (*unidirectional*).

Let me describe how the FX-100, for example, prints a typical program listing. First, it receives and prints a line (say, 50 characters), moving the print head from left to right, stopping at the end of the line. Then, it receives the command to print the next line (say, 70 characters). It moves the print head to the seventieth

column, stops, advances the paper to the next line, and prints backward from right to left. If the next line is indented (mine often are), it goes through the same sequence of stopping, moving the head, and advancing the paper once again.

But stopping, starting, moving paper, and starting again all take time. A lot of time compared to the actual printing time. Printers like the Prowriter, on the other hand, contain an internal buffer which they use intelligently. After printing a 50-character line, it checks to see where the right end of the next line needs to be and automatically continues to move the head to that position. One stop-and-start sequence eliminated. The results? See for yourself in the following chart, which records the time it took for three different printers to print the same moderate-length program listing:

Printer	Rated Speed (cps)	Time Required	Approx. Actual Speed (cps)
Decwriter	120	6 min 30 secs	110
Prowriter	120	7 min 45 secs	90
FX-100	160	7 min 30 secs	95

Oh, yes. Did I forget to mention that the Decwriter has no logic-seeking and prints unidirectionally only? That's a lot of stopping and starting. Sometimes raw power can accomplish what "logic" can't.

Throughput: True Speed

Well, I would like to report that we ran out and bought 30 or 40 different printers and tested them, too, just so I could bring you a full comparison chart. But our budget at OSS won't stretch that far.

I did, however, go to several dealers and informally time the speed of various printers. Since I had a couple of reference points (the speeds of the Prowriter and FX-100), it wasn't too hard to get a fair idea of true *throughput* figures: the printing speeds they could actually sustain.

Then I discovered another trick used by a few manufacturers. Many printers are capable of two or three (or more) character widths or fonts (typically 10, 12, and 17 characters per inch). It seems to me that at least a few printers are rated only for their smallest (and hardest to read) fonts.

Luckily we had an understanding dealer who allowed us to "trade up" our FX-100. And what printer did we then buy? Actually, we ended up buying two.

Because of our need for a printer capable of using the vast library of MX-80-compatible software, we got an Epson MX-100 (simply a wide-carriage MX-80). We have been very happy with it, though I am sure any of several MX-80-

compatible printers would have done as well. True, the MX-80 is slow. But its throughput rate seems to be around 50 to 60 cps, which is respectable compared to its rated speed.

Because we needed speed, though, we disregarded MX-80 compatibility for our other new printer, an Okidata 2350 (the model number seems to reflect its retail price). It is rated at over 300 cps and surprised us by performing our little speed test in 1 minute 55 seconds, for a throughput rate of over 360 cps. However, sometimes it gets too hot while printing long listings and stops to wait for the head to cool off. Even so, it probably has a throughput rate of 300 cps or more.

So, did you learn anything from our experiences? I sincerely hope so.

When shopping for a printer, ask to see a demonstration of its speed. Many printers perform better with uniform-length lines (such as those produced by a word processor), so ask to see a program listing also. And make your own time trials.

Judge the print quality for yourself. Ask about replacement ribbon costs. (We found one printer that worked only with carbon ribbons. \$\$\$\$! But if you need good print quality, it might be worth it.)

Above all, be certain a particular printer is compatible with your computer *and* software. Few things are worse than saving \$50 on a printer only to find out you have to spend another \$100 because your current word processor isn't compatible with your new printer.

Of Memory And Machines

We've received a few letters recently on seemingly different subjects, but which all relate to what is obviously some confusion and uncertainty about the Atari XL computers. Let's address these letters and, at the same time, shed some light on the workings of these little gems.

First, Jacqueline Patton of San Antonio, Texas, asks whether she is "stuck with a problem computer [1200XL] and an unreliable disk drive [Atari 1050]." We'll discuss the 1200XL's compatibility problems in a moment. First, a few words about the 1050.

I have not personally observed the 1050 to be any more or less reliable than any other drive on the market. Disk drives, in general, tend to be like automobiles: Sometimes you get one which goes 100,000 miles with no maintenance, and sometimes you get a lemon, but most often you get one which will last a reasonable time with reasonable care and regular checkups. This is not surprising: Disk drives and cars are both mechanical nightmares, subject to extremely close manufacturing tolerances and acute material stresses.

If the 1050 has a problem, it may be simply that it cannot read all of the more strangely protected software disks that are flooding the market. There are antipiracy measures in use today that try the limits of many drives and their controllers. Yet most programs will load fine on any good Atari-compatible drive, including the 1050.

My objections to the 1050 are centered around only one point: Although every other Atari-compatible drive manufacturer has complied with the Percom-standard double-density format (derived in turn from Atari's defunct double-density 815 drive), only Atari chose to be different. Further, Atari's method gives you a maximum of 128K bytes per disk. The others get 180K bytes. There is no excuse for this. It results from Atari's typical blindness when it comes to outside vendors.

All this does not mean the 1050 is no good. It just means that, on a bytes per dollar basis, it is overpriced.

Use Your Options

Another letter, from Shahid Ahmal of London, England, was actually a complaint to OSS about the fact that some programs (including our disk-based MAC/65) would not load and run properly on his 800XL. The problem is that these programs require you to remove the BASIC cartridge before booting up—impossible on the 800XL and 600XL, since the BASIC "cartridge" is built into the newer computers as a standard feature. His solution was to write a program which switched off the built-in BASIC, changed RAMTOP, and closed and reopened the screen driver.

Whew! I am impressed. Doing all that in the proper order is not easy. But there really is a much simpler way.

This discussion applies only to disks containing programs which *do not* use Atari BASIC. Obviously, such things as assemblers, compilers, and utility programs fit this category. Not so obviously, many game disks will not run if Atari BASIC is present. In any case, if you own an 800XL (or, I assume, a 600XL with expanded memory), and the directions for a disk or program tell you to remove your BASIC cartridge, try this:

Turn on power to all devices *except* the computer. Insert the disk you wish to boot. Push and hold down the OPTION button. Turn on the computer's power. When the disk starts to load, you can release the OPTION button.

This has the effect of disabling the built-in BASIC. Atari's manuals tell you all this. But they don't emphasize enough that you should try this with any disk/program if it otherwise doesn't work. And they don't tell you about the

OPTION button when used with the *Translator Disk*. "What's that?" Glad you asked . . .

I have mentioned the *Atari Translator Disks* before in this column, but only part of what I'll add is repetition.

If you own an Atari 1200XL, 800XL, or expanded 600 XL with a disk drive, run—do not walk—to your nearest Atari users' group and purchase (usually for about \$10) the pair of *Atari Translator Disks*. (You may still be able to get them from Atari directly.)

The instructions tell you to boot the version A disk first, wait for it to give you a message, insert your otherwise unbootable disk, and push the SELECT key. If that doesn't work, you are supposed to try the version B disk. (Both disks actually load an old Atari 800-style operating system into your XL machine's memory, thus hopefully assuring compatibility with programs that rely on the older operating system.)

What the instructions don't say is that you may also need to hold down the OPTION button. Why? Because otherwise, good old Atari BASIC is still there, messing up the memory address space.

Six Ways To Boot

There are, then, no less than six ways to try booting a disk on an XL machine: with or without holding down the OPTION button alone or in combination with either of the two *Translator Disks*. This sounds like a real pain, but once you find the method that works with a given disk, you can write it down for future reference.

I should note that all of these methods still result in compatibility with only about 97 percent of all software (85 percent of heavily protected software). Is there anything you can do if your favorite piece of software won't boot using any of these methods? Yes, two things.

First, you can write, phone, telex, or otherwise cajole and threaten the software manufacturer. I have said before, and I am sure I will go hoarse saying again, that I believe the responsibility for the lack of compatibility does *not* rest with Atari. No other manufacturer has ever produced a series of computers with as many changes and improvements as the XL line and yet maintained as much compatibility as has Atari.

Second, you can try one of the commercial translator programs. I am aware of two at this time: *XL BOSS* from Allen Macroware and *XL FIX* from Computer Software Services. I have used neither, so I cannot comment on them. However, I recommend that to avoid unnecessary expense you should certainly seek verification from these manufacturers that the particular software package you want to use will work correctly with their product.

INSIGHT: Atari

Wilkinson

As I promised, this month will be spent answering more letters. Some of the topics I will discuss here have been requested many times; others are unique queries that provide an insight into the workings of your Atari. I think they are all interesting questions.

Before starting on the questions, though, I have a bit of news that can't wait: Microbits (Albany, Oregon) is currently developing both a parallel floppy disk drive and a hard disk system for the 800XL. Preliminary speed measurements indicate that we may be able to read/write over 40,000 bytes per second to and from the disk. Imagine being able to load any of your favorite games from disk in half a second or so. Presumably, you would use the parallel floppy to back up the hard disk. Since even a five-megabyte disk (small by today's standards) takes 25 double-density floppies to back up, anything Microbits does to enhance the speed or density of the floppy will be appreciated.

Microbits has not announced any delivery dates yet (in fact, they haven't even finished development, so they can't deliver anything), but I think you should ask your local dealer to get all the information he can as soon as he can. Just think of the possibilities for graphics applications (do you realize that you could load five or six graphics mode 15 pictures per second this way? Or how about windows?).

Phase Errors

Michael Richardson, of Plattsburgh, New York, used the machine language graphics routines printed in this column in 1982 as the basis for a set of his own routines. He ran up against an unexpected error with the Atari *Assembler Editor* cartridge. Although he did not provide a complete listing, I will present what I believe is a correct excerpt here:

```
10      *= $600 ; (or any other good location)
20 DRIVE = FNAME+1 ; see below
30 ;
40      LDA DRIVE ; looks reasonable, doesn't it?
...
99 FNAME .BYTE "D1 :ANYNAME."
```

Now that tiny segment of code certainly looks innocuous, doesn't it? But when you try to assemble it, it gives you an ERROR 13, a "phase" error. Why?

Before answering the question, let's consider what would happen if we replaced line 40 with:

```
40      LDA FNAME
```

Do you know what will happen? Can you guess? Believe it or not, you will *not* get a phase error from the *Assembler Editor* cartridge.

Let's take this step by step. Remember that good old ASMED (if you will pardon my inventing an acronym for *ASseMbler EDitor*) is a two-pass assembler. On the first pass, ASMED tries to assemble LDA FNAME and discovers that FNAME has not been defined yet. "That's okay," says ASMED to itself, "I'll just assume that FNAME will be defined later as a non-zero page location. I'll reserve three bytes for this LDA instruction." Well, lo and not-too-surprisingly behold, FNAME is indeed defined later, and it is indeed not a zero page location. Thus, on the second pass through the source code, ASMED generates a three-byte LDA instruction (both in the listing and in the object code). Pass 1 and pass 2 have agreed on how much code to generate. *Voilà*, no phase errors.

What happens, though, when ASMED tries to assemble our original line 40, LDA DRIVE? Well, ASMED is smart (just how smart we will see in a moment), but it's not exactly all-powerful. When it encountered the line DRIVE = FNAME+1, it said to itself, "Aha! FNAME is

undefined. But since it is used in an expression, I must give it a value for now. Hmm. Why not give it a value of zero?"

Why not? Because then FNAME+1 is evaluated by ASMED as 0+1, and DRIVE is given a value of 1. ASMED is *not* smart enough to realize that DRIVE should be considered undefined along with FNAME.

The consequence? During pass 1 of the assembly, ASMED sees LDA DRIVE as being equivalent to LDA \$0001, a zero page reference which thus requires only two bytes of memory. But—you saw this coming, didn't you—by the time ASMED gets to LDA DRIVE on pass 2, FNAME has been defined and so DRIVE gets a value of other than one (presumably \$06xx in our little example). "Okay," says ASMED, "I'll generate three bytes for the LDA." Oops! Phase error!

Before discussing the fix for this problem, I would like to point out that many (if not all) of the other assemblers available for the Atari would also produce a phase error here. More interestingly, some (many? I haven't had a chance to try them all) would probably produce a phase error even on our other example, where we coded LDA FNAME. If so, it is because they treat undefined labels as having a value of zero, and thus reserve space for only a two-byte instruction on pass 1. The situation gets even stickier with forward referenced and/or undefined macro parameters, as implemented in the various macro assemblers available.

Anyway, what is the fix? Well, my favorite rule is simple: *Never* use a label until *after* you have defined it. I can't think of any occasion where this rule will get you in trouble. I can think of lots of ways that ignoring it can cause strange programming problems. My suggestion for the code in question would be to simply rearrange it, thus:

```
10      *= $600 ; (or any other good location)
20 FNAME .BYTE "D1 :ANYNAME.*"
30 DRIVE = FNAME+1 ; guaranteed to be defined
   now
40 ;
...
...
99      LDA DRIVE ; always three bytes now!
...

```

Give Me Room

Matthew Ratcliff, of St. Louis, Missouri, sent me a very complete listing of a program he calls "GTIA TEXTWRITER" along with some fairly thorny problems. Without repeating the actual questions, I think I can safely say they should all be lumped into the category of assembling relatively large programs on an Atari computer. Since many people (including Ratcliff) are still

using ASMED, let's begin with a look at how ASMED uses memory.

Much has been written (here and elsewhere) about how Atari BASIC allocates memory, but I can't remember ever seeing a good description of how ASMED slices up your hard-earned RAM. Shall we rectify that?

First, because ASMED was written primarily by one of the members of the Atari BASIC team (Kathleen O'Brien, and in less than three months), it is not surprising that ASMED shares many of BASIC's allocation techniques. In fact, those of you familiar with BASIC's use of the memory pointers at \$80 through \$92 would be right at home if you looked at ASMED's source code. There are, however, some major differences.

Just as BASIC has to juggle the several parts of your program (variable name table, the tokenized program, arrays, etc.), so must ASMED find places for its needed components. While you are using just the editor, this task is simple: No tokenizing takes place, no variable name or variable valuable tables are built—just straightforward expands, contracts, and inserts of your source code lines.

When you assemble, though, ASMED must find a place to put your symbol table (all the labels used in your program and what their values are, etc.). For its own convenience, ASMED simply places the symbol table in memory directly following your source code. Object code is easier: ASMED puts your object code where you tell it to. If you are assembling directly to memory, ASMED puts it in memory exactly where your *= directives tell it to.

I spot some potential trouble with that last part, don't you? But let's look at what ASMED can tell us about its usage of memory: Probably the most overlooked tool in the ASMED user's reach is the SIZE command. This is roughly the equivalent of BASIC's PRINT FRE(0). When you use SIZE, you are presented with three hexadecimal numbers. The first is the lowest non-zero page RAM being used by ASMED. The second is the current top-of-the-program source code in memory. (Even if you have no program in memory, ASMED has some fixed overhead, so this number never equals the first one.) The third hex number gives you the top of the memory which ASMED will use. Not surprisingly, the first and third numbers are derived from the Atari OS locations LOMEM (at \$02E7) and HIMEM (at \$02E5).

Let's take a hypothetical situation (which might really occur if you used a 16K machine with a cassette recorder) where you type SIZE and ASMED responds with:

0700 321C 3C1F

What does this display tell you? It tells me that this person may be in trouble. He has only \$0A03 (2563 decimal) bytes left for his symbol table when he assembles this program. Depending on the size and number of his labels, that may or may not be enough space. But that's only the first problem.

Where is the object code going to go? Aside from poor, overworked page 6 (\$0600 to \$06FF), there just *isn't* any memory free (and page 6 probably isn't big enough to hold the output from this assembly, anyway). What to do? Well, the obvious answer is to assemble your object code directly to the tape recorder. You do that simply by giving the command:

ASM „#C:

to ASMED. Then you can use NEW, check memory with SIZE again, and LOAD the object code back in memory, ready to debug it. Not bad. Time-consuming, but it works.

Or does it? Many people complain that after producing an object tape they cannot reload it successfully (usually, they get an ERROR 138, timeout). Why? Simply because ASMED turns on the cassette recorder at the beginning of pass 1, even though it may be a minute or two before pass 2 writes anything to the tape. Also, if you are producing a listing, the time taken to write the tape increases to the point where other start/stop errors are possible. There is no total fix for these problems, but here are some suggestions which might help.

First, do your assembly twice, once for the object code and once for the listing. During the object code assembly, turn off the listing (by using .OPT NOLIST as, say, line 1). Before starting the assembly, zero your tape counter. Then, as the object code is assembled to cassette, listen in (turn up the volume on your television). When you hear the first burst of data being sent to the cassette (near the beginning of pass 2 of the assembly), note the value of the tape counter. Then, to reload the object tape, rewind the tape to about five to ten seconds ahead of the counter value you noted. And that's about as good as you can do using ASMED with a cassette recorder.

Before going on, I'd like to discuss a point I sidestepped a couple of paragraphs ago. I noted that the SIZE command gave the memory used by ASMED (exclusive of symbol table space). Perhaps not obvious to many first-time users of ASMED is that you may *not* direct object code (via *) to memory anywhere between those first and second numbers. (And you'd better leave a healthy hunk alone above the second number for the symbol table.)

What happens if you don't follow this rule?

Typically, you find that your object code tries to share space with your source. Bye-bye, source. Or, worse, you may find the object code sitting on top of the symbol table. This can cause some extremely bizarre symptoms. I have seen ASMED start spitting out hundreds of errors for a single line when this happened.

Despite the fact that ASMED is one of the most bug-free programs I have ever encountered, it has a few very bad design flaws. And as we just noted, one of them is that it will assemble code right on top of memory it is using for other purposes.

However, for the disk user with 40K or more of RAM, ASMED presents no real problems if used properly. Since both the source code *and* the object code may be on the disk, the only real limitations are the sizes of the files. Obviously, the object file can be loaded in after giving a NEW command, so it need only fit between the second and third numbers given when the SIZE command is used.

But what about the source file? At first glance, it might appear that your source file is limited to what can be edited in memory. Not so! Albeit tedious, there is a way to assemble very large source files with ASMED. Simply edit the source code in pieces, none larger than ASMED's buffer space. Then, when all are ready, use the *append* capability of Atari DOS's option C to append one file after another to the first piece of the source. (*Please* do this on a copy of your master disk. It's very easy to make a mistake and append in the wrong direction.) Now you can assemble this giant source file.

There are, of course, some real disadvantages with doing things this way. The biggest of these is obvious: What happens when you get an assembly error in the middle of the fourth of the appended files? You have to edit that file and then go through the backup and append process all over again. Another problem is simply the speed of ASMED. If you expect to assemble 16K of *object* code, even without a listing to the printer, you might as well go out to a movie while you wait. A double feature. Finally, ASMED's extravagant use of zero page memory (leaving you, the programmer, only about 32 bytes) can be a real killer with large programs.

Well, we've wandered a little off the original track here, but it's all been germane to the problems of assembling large programs on your Atari. Is there a general solution to these problems? Several, if you have a disk drive. What are they? Just a nice selection of other assemblers.

ASMED is a usable introduction to machine language programming, but it is (after all) only 8K bytes long, and a lot of features had to be pared to make it fit. So when it begins to grate

on your nerves, get rid of it. What do you get instead?

Since my company (OSS) produces *MAC/65* (also a cartridge-based assembler, editor, and debugger), any answer I give is bound to be prejudiced. So I will simply tell you to go out and compare the prices, features, and speeds of the various assemblers available. You might, for instance, consult *The Book of Atari Software, 1984*, from either the Book Company or Addison-Wesley, which describes several assemblers and gives comparison charts. The advantage of getting a second assembler is that you now know what parts of *ASMED* you did *not* like, and you can look for assemblers that fix these areas.

16 Megabytes?

The topic heading here does not refer to any secret projects going on behind closed doors. Rather, I have been asked (more times than I can count) about the 16-bit version of the 6502 which has been developed by the Western Design Center (of Mesa, Arizona). I believe it is designated as the 65816, and is purported to be faster than a Motorola 68000 in many operations and capable of addressing 16 megabytes of memory. The question I am asked is fairly obvious: "Can I put this chip in my Atari and address 16 megabytes and make BASIC run faster and . . . ?" The answer is simple: *no*.

I can't let an answer like that sit around naked, so let's see if we can't flesh it out a bit. First, in order to address 16 megabytes, you have to *have* 16 megabytes. Have you seen any 800XLs with a lot of spare RAM floating around lately? Further, addressing 16 megabytes means you *must* have 24 address lines. (The 16 address lines in your Atari computer can access only 64K.) There simply isn't any place provided on the Atari circuit boards for such an expanded address bus.

Now, at least one version of the 65816 is purported to be pin-compatible with existing 6502s. If this is wrong, I apologize. I admit I am repeating what I have been told. Presuming this to be true, though, it may barely be possible to imagine an expansion box for an 800XL which can properly decode some sort of I/O signal to "bank" in additional RAM. I suspect, though, that the pin-compatible version may be so compatible that it limits you to 64K of memory.

So far, however, this highly hypothetical discussion has assumed that the chip will be compatible enough (with a 6502) to fool the rest of an 800XL's circuitry. I'm not convinced that this will prove to be true. Why? Because the 65C02 (which, you may or may not recall, is a CMOS version of the 6502 which adds a few—still all 8-bit—instructions and capabilities) does

not work in an 800XL. Even though it works *great* in older Atari 800s.

I am not sure why the 65C02 is incompatible with the 800XL, but I have been told it is because Atari started using a custom version of the 6502 in its newer machines. (The story is that the newer CPU is the same one found in the 2600 game machines, and it has one or two pins used differently.) In any case, the problems with the 65C02 cause me to doubt that the 65816 will enjoy a better fate.

Last, let us assume that you really can plunk a 65816 down into the middle of your 800XL. Will it do you any good? Not unless you are a heavyweight in machine language. Compatible means just that: It executes all standard 8-bit 6502 instructions in the same old way. And where are you going to get any of the new 16-bit instructions from? I dunno. It is extremely doubtful that any major software vendor will be able to justify the expense of developing programs which use the 65816 in an Atari, since using the chip involves doing nasty things to your computer that very, very few users are willing to try.

And there you have it. I hope I am wrong about much of the above, solely for my own personal satisfaction with such a 16-bit machine. But—sigh—I am probably mostly right. (But what if . . . nah . . . it couldn't happen.)

Super Cart

Copy Atari 400/800/XL Series Cartridges to Disk and run them from a Menu

ATARI CARTRIDGE-TO-DISK COPY SYSTEM \$69.95

Supercart lets you copy ANY cartridge for the Atari 400/800/XL Series to diskette, and thereafter run it from your disk drive. Enjoy the convenience of selecting your favorite games from a "menu screen" rather than swapping cartridges in and out of your computer. Each cartridge copied by Supercart functions exactly like the original. Supercart includes:

- **DISKETTE:**
 - **COPY PROGRAM** - Copies the cartridge to a diskette (up to 9 cartridges will fit on one disk.)
 - **MENU PROGRAM** - Automatically runs and displays a menu prompting user for a ONE keystroke selection of any cartridge on the disk.
- **CARTRIDGE:**
 - "Tricks" the computer into thinking that the original "copy protected" cartridge has been inserted.

To date there have been no problems duplicating and running all of the protected cartridges that we know of. However, FRONTRUNNER cannot guarantee the operation of all future cartridges. Supercart is user-friendly and simple to use and requires no modifications of your hardware. **PIRATES TAKE NOTE:** SUPERCART is not intended for illegal copying and/or distribution of copyrighted software. . . Sorry!!!

SYSTEM REQUIREMENTS:

Atari 400/800 or XL Series Computer / 48K Memory / One Disk Drive

Available at your computer store or direct from FRONTRUNNER. DEALER INQUIRIES ENCOURAGED.

TOLL FREE ORDER LINE: (24 Hrs.) 1-800-543-5500 (In Nevada or for questions Call: (702) 796-4600)

Personal checks allow 2-3 weeks to clear.
Include \$3.50 (\$7.50 Foreign orders) for shipping.

FRONTRUNNER COMPUTER INDUSTRIES
316 California Ave., Suite #712, Reno, Nevada 89509 - (702) 796-4600
Others Make Claims. . . SUPERCART makes copies!!!

ATARI is a trademark of Warner Communications, Inc.

Atari Touch
Tablet

\$39⁹⁵

COMPUTER

MME

CENTER

Atariwriter

\$37⁹⁵

<p><small>\$10 shipping</small></p> <p>Atari 1020 Printer Plotter \$65⁹⁵</p> <p>Indus GT \$295⁹⁵</p> <p>Panasonic Printer \$229⁹⁵ <small>KXP1090, Atari-Commodore 120 cps</small></p> <p>Atari 1050 Disk Drive \$224⁹⁵</p> <p>Atari 1010 Recorder \$49⁹⁵</p> <p>FREE CATALOGUE - Mail \$1.00 for postage and handling to: MME, PO Box 9009, Fairfield, OH 45014</p>	<p style="text-align: center;">Software</p> <p>Pac Man, Eastern Front, Centipede, Qix, Defender (For Atari) \$12⁹⁵ each</p> <p>Maxell MD1 Box 10 \$17⁹⁵</p> <p>Easy Script Commodore \$39⁹⁵</p> <p>Compuserve Starter Kit \$29⁹⁵</p> <p>Micro Filter (MPP) Atari \$38⁹⁵</p>
--	--

Postage Handling < Hardware \$10 - Foreign \$15 - Software \$4

VISA-Mastercard C.O.D. Money Order-Cash 5% Ohio Sales Tax
CALL 1-800-543-5500 Ask For Operator 525
INQUIRIES 513-874-2084

INSIGHT: Atari

Bill Wilkinson

I have almost worked my way through my backlog of letters, so I will once again appeal to all of you to keep those cards and letters coming. Since it can sometimes take quite a while for a letter sent to *COMPUTE!*'s editorial offices to wend its way to me, I have decided to give you an address where you can write me directly:

Bill Wilkinson
c/o OSS
P.O. Box 710352
San Jose, CA 95171-0352

Before I start answering questions this month, I would like to talk a little about the future of Atari.

Right From The Source

I had the rare privilege to attend the meeting of the San Leandro (California) Atari User Group on the evening that Leonard Tramiel agreed to come and answer questions.

I hope the name Tramiel is familiar to all Atari owners by now. Jack Tramiel, the founder and former leader of Commodore, bought Atari from Warner Communications in July. Leonard, Jack's son, is now head of software at Atari. And though I am sure some favoritism was involved in choosing him for the position, I think it was probably an excellent appointment.

Leonard Tramiel is an articulate, humorous, open, and opinionated person. He endeared himself to me when he espoused one of my favorite opinions: The IBM PC is an eight-bit machine, and the Apple Macintosh is a 16-bit machine, and no amount of marketing ballyhoo is going to change that. (We are referring to the fact that the width of the data path to the Central Processing Unit (CPU) controls processing speed as much as, if not more than, the speed of register operations. Whew! Got that? There will be a quiz on Monday.)

Anyway, while Leonard was extremely careful to avoid divulging any technical details about future Atari computers, he went a long way toward reassuring many listeners (for example, me)

that Atari in general (and Leonard Tramiel in particular) knows what it is doing and where it is going. By the time you read this, the Winter Consumer Electronics Show (CES) in Las Vegas will be underway. And we're expecting to see the introduction of 16-bit and 32-bit Atari computers.

However, I also came away with the feeling that Atari will not abandon the eight-bit, 6502-based market for some time to come. In particular, Leonard stated emphatically several times that the 800XL would undergo only those modifications which would make it "both less expensive and more reliable."

Preserving Atari Loyalty

Possibly Leonard missed his calling: As a public relations person he did an outstanding job. I didn't take a formal poll, but I believe the impression he left on the audience was in the range of 90 to 95 percent positive. If there were any real negatives, it was regarding his stand that he wouldn't guarantee that current Atari peripherals would work on the new machines.

The attitude of some in the audience was, "Well, if I can't use my peripherals on the new machines, I am going to look at all computers instead of just Atari's." That's a reasonable attitude, but the response was just as rational: "If Atari can't convince you to buy the new machines on their merits and prices alone, then we don't know what we are doing." And finally, my view is that—with the possible exception of printers—there are very, very few Atari peripherals that I would want on a new, super-duper computer. (Who wants to talk to a disk drive at 19200 baud? Who really likes the kludge that became the 850?)

In summary, then, I have a better feeling about the future of Atari than I have had in a year or more now: to the point that our company, OSS, is continuing with plans for more and new Atari-compatible products. I will withhold judgment of the new machines until I see their software (*Please* give us an operating system! Not CP/M, MS-DOS, or Apple or Commodore style!), but with Leonard Tramiel's

leadership I have some hopes in that direction, also.

Where It's At

I've received a few letters in recent weeks asking if there is a good list of important memory locations for Atari computers. Oh, come now, COMPUTE!. Can it be that you are not advertising your 1983 book *Mapping the Atari*? To my knowledge, this is the one and only complete memory map of Atari computers. Further, it is much more than a memory map. It gives example programs, discusses which system routines will use and/or change certain locations, and much, much more. And yet there are readers of this magazine who are not aware of this book! How can that be?

Well, to be fair, the cover of *Mapping the Atari* does state that it is intended for owners of Atari 400 and 800 models. However, the people who wrote me own either 1200XLs or 800XLs. Does that matter? Not really.

More than 99 percent of the significant memory locations are the same in *all* Atari computers: 400, 800, 1200XL, 600XL, 800XL. Notice that I did qualify that just a little. Just what is a *significant* memory location?

Sidetrack: If you have been reading this column for any time at all, you know that I feel that the compatibility problems which many software vendors suffered when the XL machines appeared are the *fault of the vendors*. Since the first documentation from Atari appeared in the marketplace, Atari made a point of specifying which memory locations would control what functions, which subroutine entry points (mainly vectors) would remain unchanged, and which parts of the operating system (OS) were subject to change. Surely, when Atari released its first revision of the OS in early 1982, you would think the vendors and authors would have been put on notice: "Hey, guys, things are subject to change, and this proves it." The reply: "Yeah, but if I know that this routine at \$D099 will save me two bytes of code, I'm gonna use it."

The only consolation I seem to get is that every other machine seems to have the same kind of problem: Apple programmers had to go back to the drawing board when the IIe and IIc arrived. Many major programs for the IBM PC simply do not run on the PC-AT. Nobody can write machine language software for Commodore computers and expect it to work on more than a single model. The list goes on.

Mapping XL Memory

Back to the memory map: Generally, if you use *Mapping the Atari* with an XL machine, you can trust most of the RAM locations that are listed. Atari did publish a set of locations that were

changed in the XL machines, but there were not many. Even the ones that did change were ones unlikely to be used: OLDROW and OLDROW moved, but the only routines that use them are FILL and DRAWTO. And even if you were to call for a FILL, you probably would do so after a PLOT, which automatically sets up OLDROW and OLDROW for you.

The ROM locations listed in the book are a bit more subject to change. As a rule of thumb, I would trust only the information about the last few bytes of a cartridge, the floating point ROMs, and \$E400 through \$E462. Also, it's a pretty sure bet that if the book mentions a difference between OS revision A and revision B when discussing a location, there will be yet another difference in the XL machines. (Example: Anybody who thinks that EOUTCH—output a character to the screen—is at an immutable location should refrain from using a machine manufactured after 1916.)

So all you XL machine owners should rush out and buy a copy of *Mapping the Atari*. And then you should write to COMPUTE! and tell them (don't ask) to publish an update, either in the form of a revised book or a low-cost appendix, for XL computers.

More No-Nos

As long as we are on the subject of only using *legal* memory locations (see how I sneaked that in?), let me respond to a couple of people who have asked a relevant question: "I have an 800XL, and I can't get it to put characters to the screen if I follow the instructions in *Machine Language for Beginners*. How can I change the program so it will work?"

When Richard Mansfield wrote that book, he was writing for Commodore, Apple, and Atari owners. And all the machines he was writing for *except Atari* have a documented entry point for a routine which will put a single character on the screen. So, for uniformity, he used an undocumented subroutine call on the Atari computers which does much the same thing. At the time he did this, that particular location had been written up several times in both the professional and amateur press, so he felt fairly safe. Ah, well, Richard, even the best of us have to be bitten once in a while.

The proper way to do any input/output (I/O) on an Atari computer is via Central Input/Output (CIO) calls. In early 1982, I wrote a series of articles on CIO calls which appeared in this column. I am not going to repeat that series, but I will give you a few pointers to get you started with CIO.

There are two things you can do if you want more info on the subject: (1) Find a library (per-

haps a user group library) with back issues of COMPUTE! (don't write the magazine; they don't have any). (2) Get your hands on a copy of the *Atari Technical Reference Manual* (it used to be \$30 from Atari customer service, but I don't know where you can get it now). The manual includes a pretty fair description of CIO along with lots and lots of other very worthwhile goodies.

The Legal Solution

Without further ado, then, let's look at how to put a character on the screen.

```
0200 IOCB0 = $0340
0210 IOCB_CMD = $0342
0220 IOCB_LEN = $0348
0230 CMDPUT = $0B
0240 CIO = $E456
0250 ;
0260 ;Enter with character in A
      register
0270 ;Routine will print it to screen
0280 ;
0290 PUTSCREEN
0295     LDX #CMDPUT
0300     STX IOCB_CMD ; request output
0310     LDX #0       ; multi-purpose.
      zero
0320     STX IOCB_LEN ; first, zero
      length
0330     STX IOCB_LEN+1 ; (both bytes)
0340     JMP CIO       ; and now X is
      channel for CIO
```

That's it. Simply put those six lines of code anywhere in your machine language program. Then, when you want to print a character on the screen, use JSR PUTSCREEN after placing the character in the A register.

In theory, you can get an error when you call CIO (a minus value in the Y register indicates this), but in practice I don't believe you will ever see one as a result of putting a character to the screen.

How, you may ask, is this any better than calling a point in the OS ROM which does the same thing? Answers: (1) This way works on all Atari computers (well . . . the 6502-based ones, at least). (2) This follows Atari's rules. If you do it this way, Atari could scramble the OS ROMs anyway they wanted, but your program would still run.

Of course, the equates at the beginning of the program fragment are the keys to the whole thing. IOCB stands for *Input/Output Control Block*. Technically, you are supposed to put the channel number times 16 in the X register and then access the appropriate IOCB via X (see below). Since the screen is always open on channel 0, I took a legitimate shortcut. Similarly, CIO is actually a vector in the OS ROMs which is guaranteed to stay in place. If you follow the rule about using the X register to access the IOCBs, you are already set up for CIO, which requires


the channel number times 16 in the X register.

Oh, yes. Normally, CIO expects to transfer an entire buffer (for example, a line of text), in which case you must give CIO the buffer address and its length. But CIO cleverly provides for situations in which you want to print only a single character: Tell CIO that the length of the buffer is zero, and it will output a single character (or input a character, but that's a topic for another time) via the A register.

And that's about it. Simple, really. Before we quit for this month, though, I would like to show you how simply that routine could be converted to output a character to *any* channel.

```
0200 IOCB0 = $0340
0210 IOCB_CMD = $0342
0220 IOCB_LEN = $0348
0230 CMDPUT = $0B
0240 CIO = $E456
0250 ;
0260 ;Enter PUTC with the character
0270 ; in the A register and the
0275 ; channel number times 16 in
      the
0280 ; X register.
0285 ;
0290 PUTC
0300     PHA           ; save character
      for a moment
0310     LDA #CMDPUT ; request output
      ...
0320     STA IOCB_CMD,X ; ... on this
      channel
0330     LDA #0
0340     STX IOCB_LEN ; now, zero
      length
0350     STX IOCB_LEN+1 ; (both bytes)
0360     PLA           ; recover the
      character
0370     JMP CIO       ; and now X is
      channel for CIO
```

Do you see the really minimal changes we made? This is one of the beauties of the Atari OS. It is so completely organized (*orthogonal* is a good computerese word for it) that it's actually easy to learn and use. Perhaps we'll do a little more of this if you would like. Write and tell me.



SuperCart™

Copy Atari 400/800/XL Series Cartridges to Disk
and run them from a Menu

ATARI CARTRIDGE-TO-DISK COPY SYSTEM \$69.95

Supercart lets you copy ANY cartridge for the Atari 400/800/XL Series to diskette, and thereafter run it from your disk drive. Enjoy the convenience of selecting your favorite games from a "menu screen" rather than swapping cartridges in and out of your computer. Each cartridge copied by Supercart functions exactly like the original. Supercart includes:

- DISKETTE with:
 - COPY PROGRAM - Copies the cartridge to a diskette (up to 9 cartridges will fit on one disk.)
 - MENU PROGRAM - Automatically runs and displays a menu prompting user for a ONE keystroke selection of any cartridge on the disk.
- CARTRIDGE:
 - "Tricks" the computer into thinking that the original "copy protected" cartridge has been inserted.

To date there have been no problems duplicating and running all of the protected cartridges that we know of. However, FRONTRUNNER cannot guarantee the operation of all future cartridges. Supercart is user-friendly and simple to use and requires no modifications of your hardware. **PIRATES TAKE NOTE: SUPERCART is not intended for illegal copying and/or distribution of copyrighted software. . . Sorry!!!**

SYSTEM REQUIREMENTS:

Atari 400/800 or XL Series Computer / 48K Memory / One Disk Drive

Available at your computer store or direct from FRONTRUNNER. DEALER INQUIRIES ENCOURAGED.

TOLL FREE ORDER LINE: (24 Hrs.) 1-800-546-4788 / In Nevada or for questions Call: (702) 795-4600

Personal checks allow 2-3 weeks to clear. M/C and VISA accepted.

Include \$3.50 (\$7.50 Foreign orders) for shipping.

FRONTRUNNER COMPUTER INDUSTRIES

316 California Ave., Suite #712, Reno, Nevada 89509 - (702) 795-4600

Others Make Claims. . . SUPERCART makes copies!!!

ATARI is a trademark of Warner Communications, Inc.

A Sequel On The Way

Roberta Williams is a perfectionist. There just isn't time or memory to put everything in *King's Quest* that she wanted. She wishes the language interpreter had a larger vocabulary, that Sir Grahame could drop objects he has picked up, and that he could be even more animated. She wishes some of the characters, like the wolf, could roam from room to room. But she says the sequel, due in February or March, will be even better.

In *King's Quest II*, Sir Grahame—who becomes King Grahame when you solve *King's Quest*—goes in search of a wife. Along the way he meets Dracula and King Neptune, and rides a

flying carpet. And, somehow, the folks at Sierra found a way to squeeze 94 rooms onto the disk. I can hardly wait.

IBM markets the PCjr version of *King's Quest* and Sierra markets versions for the IBM PC, Apple IIc, Apple IIe, and Tandy 1000 (a new computer scheduled for release in January 1985). All versions cost \$49 and require 128K of memory and a disk drive. In addition, the PC version runs on most IBM compatibles. (When the PC version is displayed on an RGB monitor, the graphics are in the standard four-color medium-resolution mode; but connect your PC to a television and you'll get the same spectacular colors as the PCjr version.) ©

INSIGHT: Atari

Bill Wilkinson

I am much gratified by the response to my decision to work harder on answering readers' questions. I have received several *very* interesting letters with both good comments and good questions. Since it is always fun to defend Atari BASIC against the outside world, let me start with a subject near and dear to my heart.

Benchmarks

Several readers have asked me why Atari BASIC compares so unfavorably to other computers on certain benchmarks. The two most commonly mentioned are the *BYTE* magazine benchmarks and the *Creative Computing* benchmark invented by David Ahl. Stan Smith, of Los Angeles, asked some very pointed questions, which I will try to answer here.

The *BYTE* benchmark is reproduced below in Atari BASIC. It is the often-mentioned "Sieve of Eratosthenes," a program which produces (and counts) prime numbers. Its primary advantage as a benchmark is that it can be implemented in virtually any language (although only with much difficulty when using Logo and its ilk). It relies

only on addition and logical choices, with very little number crunching.

```
10 DIM N$(8192)
20 N$="0":N$(8192)="0":N$(2,8192)=N$
30 FOR I=1 TO 8192:IF N$(I,I)=""
  1" THEN 60
40 PRIME=I+I+1: CNT= CNT+1: K=I
50 K=K+PRIME: IF K<8193 THEN N$(K,K)="1":GOTO 50
60 NEXT I
70 PRINT CNT : REM BETTER PRINT
  1899!!!
```

An aside: If you have seen the *BYTE* original and are puzzled by my changes, be aware of three things: (1) I had to use a string because there is not enough room for an array of 8192 elements. (2) The math was modified very slightly to accommodate the fact that string indices start at one, instead of zero. (3) Multiple statements per line simplify the original somewhat.

Anyway, why is Atari BASIC so slow (317 seconds versus, for example, the IBM PC at 194 seconds)? Primarily for three reasons. First, note

all the numbers in this listing, which must be treated as integers. Line numbers and indices are always kept and calculated as floating-point numbers, but all must be converted to integers before being used. (You simply can't GOTO line 137.38, can you?) And, sigh, the routine in the Atari Operating System ROMs which converts numbers to integers is incredibly slow (in fact, it is the only floating-point routine we modified when we produced BASIC A+ and BASIC XL).

Second, Atari BASIC performs FOR-NEXT loops by remembering the line number of the FOR statement. Then, when NEXT is encountered, BASIC must search for the FOR line, just as if a GOTO had been used. (Other BASICs remember the actual memory address of the FOR statement. Faster, but less flexible. Atari BASIC allows you to STOP in the middle of a loop, change the program, and continue, something no other home computer BASIC allows. (This—among many other things—is in direct opposition to *Consumer Reports'* claim that Atari BASIC is hard for beginners.)

Third, if you type in and use this listing as shown, you are paying almost a 50 percent penalty in speed, thanks to Atari's screen DMA and Vertical Blank Interrupts taking up a significant portion of the processing time. The simple addition of the following two lines will improve the time for this little test to 211 seconds:

```
5 POKE 54286,0 : POKE 54272,0
65 POKE 54286,64
```

All of a sudden, Atari BASIC isn't even *near* the bottom of the list. And, yet, there is more we can do to improve the machine's performance. As many have suggested, you can install the Newell Fastchip, a replacement for the floating-point routines built into your computer (available from many dealers, produced by Newell Industries of Plano, Texas).

Or you can change to another BASIC. Obviously, there is Atari's Microsoft BASIC. It produces results very close to those of Applesoft; but it, too, can be improved by turning off screen DMA, etc. And there is OSS's own BASIC XL. Using a combination of clever programming and a Fastchip, the BASIC XL program below will count up all those prime numbers in 58.5 seconds, about three times as fast as Microsoft BASIC on an IBM PC can do it. 'Nuff said. (Except a P.S.: The Set 3 in line 10 requests zero-time FOR loops, something not available in many BASICs, which alone accounts for about 20 seconds worth of improvement.)

```
10 FAST: POKE 54286,0: POKE 542
72,0: SET 3,1: DIM N$(8192):
N=ADR(N$)
```

```
30 FOR I=0 TO 8191
50 IF NOT PEEK(N+I) THEN PRIME=
I+I+3: CNT=CNT+1: FOR K=I+PR
IME TO 8191 STEP PRIME: POKE
N+K,1: NEXT K
60 NEXT I
70 POKE 54286,64: POKE 559,34:
PRINT CNT
```

Measures Of Accuracy

The Ahl benchmark is listed below. It purports to measure both accuracy and number-crunching ability. It does neither very well. Still, we have to ask why Atari BASIC is near dead last in its rankings, requiring 6 minutes and 45 seconds to complete the test.

```
10 FOR N=1 TO 100: A=N
20 FOR I=1 TO 10: A=SQR(A): R=R
+RND(0): NEXT I
30 FOR I=1 TO 10: A=A^2: R=R+RN
D(0): NEXT I
40 S=S+A: NEXT N
50 PRINT "ACCURACY="; ABS(1010-
S/5), "RANDOM="; ABS(1000-R)
```

The culprit here (in terms of time-wasting) is line 30, with its $A=A^2$. Atari BASIC, in common with most small computer BASICs, calculates powers according to a formula:

$$x^y = \exp(y * \log(x))$$

where $\log()$ is the natural logarithm function and $\exp()$ is the exponent-of-e function.

If you don't understand that, don't worry about it. The point is that the calculation of such a simple thing as a number to the second power involves the calculation of a logarithm and an exponentiation. And why is that so bad? Simply because the floating-point routines in the Atari OS ROMs are too slow. Again, the solution is to install the Newell Fastchip and/or turn off DMA and VBI (as outlined above).

I am indebted to Clyde Spencer, one of the founders of the Bay Area Atari Users Group (one of the oldest), for supplying me with a most surprising figure. Spencer reports that, using the Fastchip and with DMA turned off, he obtained a timing of 1 minute 38 seconds, a very respectable (albeit not record-shattering) performance. I still wouldn't use my Atari for advanced scientific applications, but it is more than adequate for most purposes.

There is a problem with the "accuracy" figures in this test, however. First, because Ahl's accuracy number is the result of 1000 simple sums, it is clearly possible that a particular machine may exhibit wildly variant results for various numbers and still show a good figure in his test. (To illustrate, assume that the SQR() function randomly tosses in an error of plus or minus

one. If it tossed in an equal number of errors, they would balance to zero. Yet choosing to make the loop just one unit shorter [FOR N=1 TO 999] might give a completely different result. To be fair, this is a very unlikely result with modern math algorithms; but, still, one never knows.) A minor change to his program would improve the testing qualities considerably:

```
40 S = S + ABS(A-N) : NEXT N
```

Do you see the difference? This method produces the sum of the errors, and doesn't fall prey to offsetting errors.

The Random Number Trap

There is no hope for the accuracy of this random number tester, though. I will quote Clyde Spencer on this matter: "If the numbers are *truly* random and *not* normally distributed, *any* difference between 0 and 1000 is possible. All you can say is that you would have a high *probability* of . . . being near zero for a perfect random number generator." The benchmark test falls into the infamous BASIC repeating-random-sequence trap.

In most BASICs, when you command a program to run, the pseudorandom generator is *always* reseeded with the *same* number. So each and every time you will get the same results, with Ahl's test. And, depending on what seed is chosen, you may get truly phenomenal results (because you happened to hit a hot spot in the generator's sequence). Now, though, try starting the generator off with a different (and randomly chosen) seed each time. What happens? The test's randomness figure wanders all over the place.

Once again, to quote Spencer, ". . . in eight tests I obtained numbers ranging from 1.6 to 24.2, with the mean being 7.02 . . ."

Finally, I would like to point out that Ahl's test penalizes small machine BASIC interpreters in yet another way: When you have 32K bytes to spend on a BASIC, one thing you do is insure that numbers to a power are performed by successive multiplications, if possible. Thus Cromemco 32K Structured BASIC (for example) performs A^2 with just one multiply. In other words, it converts A^2 to $A*A$. If you manually substitute that same form in Ahl's program, the times for almost all of the smaller and less expensive machines will improve dramatically. (Surprisingly, though, the accuracy figures may not change. After all, the original version may have had offsetting errors.) Of course, if you need to use noninteger powers in your programs, this comment doesn't apply, and the benchmark's results are a bit more meaningful for you.

Well, what does all this long-winded discussion boil down to? Two simple points: (1) Al-

ways presume that a benchmark program is worth slightly less than the paper it is printed on. (2) If you want to do number crunching on your Atari computer (against my best advice), go out and buy the Newell Fastchip. (And it won't hurt to try some other languages.)

HELP? HELP!

Besides noting that GRAPHICS 15 on the XL machines is easily accessible (it's equivalent to mode 7½ on older machines), Mark Butler, of Antioch, California, asked for some information about the HELP key.

Simply put, pushing the HELP key on an XL machine causes an interrupt (I'm not sure which one) that, in turn, causes the Operating System to set a HELP flag. The magic location is \$2DC, 732 decimal. Pushing HELP, either alone or in combination with CONTROL or SHIFT, forces the OS to put a value here, as shown below:

Key(s) Pressed	Value in \$2DC (732)
HELP alone	\$11 (17 decimal)
CONTROL+HELP	\$91 (145)
SHIFT+HELP	\$41 (65)

To use \$2DC, you must POKE it back to zero after you have decided that someone needs HELP which you are going to act on.

Butler also requested a program which would, for example, print out an error message for the last BASIC error number when the HELP key is pressed. While not a *really* difficult project, it is a bit too heavy for this column. On the other hand, it would be trivial to add a HELP capability to many BASIC programs. Why not try it?

As long as we are on this subject, I would like to also note the effects of the 1200XL's function keys on another memory location, \$2F2 (754 decimal). The various possible values are listed below. Note that CONTROL used with a function key is not generally accessible after keyboard input, since these combinations have special meanings to the OS and the editor handler. We will thus ignore them here.

Key(s) Pressed	Value in \$2F2 (754)
F1 alone	\$03 (3 decimal)
SHIFT+F1	\$43 (67)
F2 alone	\$04 (4)
SHIFT+F2	\$44 (68)
F3 alone	\$13 (19)
SHIFT+F3	\$53 (83)
F4 alone	\$14 (20)
SHIFT+F4	\$54 (84)

Too bad all machines don't have function keys, isn't it?

Cassettes And The XL Machines

Guy Servais, of Norfolk, Virginia, was one of several who I inadvertently ignored when I discussed holding down the OPTION key while

booting an 800XL computer. My apologies for slighting you cassette owners.

Still, my general comments apply: If you purchase a cassette program which includes instructions telling you to remove your BASIC cartridge, you *must* hold down the OPTION key while booting that cassette. The kicker here, though, is that you must *also* hold down the START button to force the boot in the first place. Under the conditions mentioned, I recommend holding down *both* buttons until you actually hear the tone on the tape being accepted by the computer.

Servais also asked me if you can "disable the built-in BASIC . . . and can type in programs written in machine language." I can only presume that he has either seen or used other brands of computers which have some sort of minimonitor which allows you to access the bits and bytes of memory. (For example, Apple II computers have a small monitor which you can get to.)

Sorry, Guy, but there ain't no such thing on an Atari computer. You have three choices:

1. Use BASIC. This isn't quite as bad as it sounds. Look at the MLX machine language loader which COMPUTE! uses. It is a good tool for entering machine language written by others.
2. Buy a cartridge-based assembler. The old

Atari Assembler Editor cartridge is often available at a substantial discount. It's not great, but it's much better than the simple monitors on other machines.

3. Buy a disk drive. This will open up a whole new vista in machine language. There are several appropriate assemblers for disk users.

Even though I have said this before, it bears repeating: The *first* peripheral you should buy is a disk drive. Only use cassette if you are desperate, and never waste your money on a printer until you have a disk.

Can You Help?

Servais mentioned one more thing in his letter which disturbed me. He is experiencing the infamous Atari BASIC editing lockup in his 800XL with the built-in BASIC. I had believed that the 800XL's BASIC had cured that problem (though it left a few other bugs lying around). Now, truthfully, I haven't used much besides BASIC XL in the last year, so I have not been aware of this problem at all.

Has anyone documented the circumstances under which lockup occurs? Please write and tell us. Once again, since BASIC is in ROM, I doubt there is a fix for the problem. But if we are aware of why and how it occurs, we may be able to warn others away from those conditions.

C

COMPUTER T-SHIRTS

FOR HOME, SCHOOL, AND OFFICE!



FREE
IN-COMPUTERS
BUMPER
STICKER

Made in U.S.A.

The **ULTIMATE** Software!
Command instant attention!
AMUST for all computer lovers!
BRIGHT GREEN (LCD) LETTERS
CUSTOM SILKSCREENED ON 50/50 BLEND
— HIGH TECH DESIGN! —
Five popular styles to choose from
Order Today! Only \$8.95 ppd
Simply select Shirt, and Color below

LET'S SEEK, PEER & POKE (#1)	White #1	Pink #2	Blue #3
TAKE A BYTE OUT OF ME (#2)	Green #4	Grey #5	Red #6
I'M USER FRIENDLY (#3)	Black #7		
HAPPINESS IS A PROGRAM THAT WORKS (#4)			
I ♥ COMPUTERS (#5)			

— CUT — SIZES S—M—L—XL

Please send me _____
Shirt _____ Colors _____ Size _____ / Shirt _____ Colors _____ Size _____
Use additional sheet if necessary

COMPUTER NOVELTY CORP
P.O. BOX 2964
FREEPORT, TEXAS 77541
Enclose \$8.95 ppd each TX Res. 6% tax

NAME _____
ADDRESS _____
CITY _____ STATE _____ ZIP _____

©CNC

WOW!



IBM PC w/Drive
\$1299.95
OKIDATA 92
\$349.95

PRINTER SPECIALS			
Juki 6100	369	Okidata 92	349
Panasonic KXP 1091	254	Okidata 93	549
Panasonic KXP 1090	199	Okidata 84	819
Silver Reed EXP 550	378	Okidata 83	524
Silver Reed EXP 500	324	Okimate 10	159
Silver Reed EXP 770	759	Epson RX80 FT	284
Prowriter 8510	329	Epson RX80	237
Nec 2050	659	Epson RX 100	394
Nec 3050	1335	Epson FX80	409
Diablo 620 API	699	Delta 10	334
Mannesman Spirit	239	Delta 15	464
Riteman Blue	279	Epson FX 100	594
		Epson LQ1500	1119
		Toshiba 1351	1239
		Gemini 10X	229
		Gemini 15X	339
		Radix 15	577
		Radix 10	496
		Brother HR15	344
		Brother HR25	584
		Brother HR35	799
		Keyboard	129
		Olympia Ro	314
		Powertype	299
		Daisywriter	784
		Teletex 1014	349

IBM

PC w/Drive	1299
PC XT w/Drive	2699
Portable	1799
PC Jr	459
Monitor Card	159
Color Card	169
IBM Monitor (GRN)	244
Hercules Graphic	
Master	309
Tecmar Captain 64K	269
AST Six Pack	219
Tallgrass 20 Meg	2395

CALL TOLL FREE
800-VIDEO84
800-441-1144

APPLE

2E w/Disk Drive	859
Macintosh	1669
Apple 2C	874
Imagewriter	486
Apple IIc	269
Add'l Drives From	99

COMMODORE

Commodore 64	184
1541 Disk Drive	204
1702 Monitor	204
MPS801 Printer	179
1526 Printer	89
1650 Modem	224

MODEMS

Hayes 1200	439
Hayes 1200B	378
Hayes 300	189
Microcom 2E	214
Access 123	364
Novation J-cat	96

MONITORS

Amdex 300 Green	114
Amdex 300 Amber	124
Color 300	234
Color 500	324
Color 600	384
Color 700	489
310 Amber	140
Taan 210	205
Zenith GRN	129

SANYO

550 S.S.	649
550 D.S.	659
555 S.S.	824
550 D.S.	959
CRT 30	99

ATARI

800 XL	114
1027	229
1050 Drive	154
Indus Drive	279

HARMONY VIDEO & COMPUTERS
2357 CONEY ISLAND AVE., BROOKLYN, NY 11223
TO ORDER CALL TOLL FREE
800-VIDEO84 OR 800-441-1144
IN NY (718) 627-1000

NEW!

Universal Parallel Graphics Interface



- Built-in self-test with status report
- Optional RAM printer buffer
- Provides virtually total emulation of Commodore printers for compatibility with popular software
- ASCII conversion, total test, Emulate & transparent mode
- Fully intelligent interface that plugs into standard Commodore printer socket
- Exclusive graphic key-match function
- Switch selectable Commodore graphics mode for Epson, StarMicronics, C.Itoh, Prowriter, Okidata, Seikosha, Banana, BMC, Panasonic, Mannesman Talley & others.

Micrografix MW-350 \$129.00
MW-302C Printer Interface also available at \$79.95




Dealer inquiries invited.

Micro World Electronix, Inc.
3333 S. Wadsworth Blvd., # C105,
Lakewood, CO 80227
(303) 987-9532 or 987-2671

INSIGHT: Atari

Bill Wilkinson

It's tidbit time again this month! I love saving up strange, exotic, or frustrating facts and then dumping them on you all at once.

Oops

Before I do anything else, though, let me quickly fix the big boo-boo in my January column or I will be inundated with threatening letters. In the *second* assembly language listing there, the one which purported to show you how to output a character to any channel, there are two lines in error. Lines 340 and 350 were given as using a STX instruction. The correct lines are as follows:

```
340 STA IOCBLEN,X ; zero both bytes
350 STA IOCBLEN+1,X ; of buffer length
```

I apologize now if I managed to destroy anyone's hard work. And since I goofed in January, it's only fair to show my ignorance this month.

Hide And Seek

The first of the tidbits this month came to me in the way of an innocent question from Roger Bocek of Campbell, California. He had stopped in at our office to pick up some software, happened to run into me, and said, "Say, I've been meaning to ask you. Why does DOS use three sectors for its boot area when it uses only about 200 bytes of boot code?"

Five minutes later, after rummaging through the listing in *Inside Atari DOS* (from COMPUTE! Books, of course), I came up with the brilliant answer: "I dunno." But I always like to find a use for everything, even my own ignorance.

As we have discussed in this column many times in the past, when you ask BASIC to do I/O (Input/Output) to or from most devices attached to your computer (particularly the disk drive), what actually happens is quite complex. BASIC interprets your request into a call to CIO (Central Input Output), which in turn determines what device you are using and vectors to the appropriate driver routine. We assume here that CIO accesses FMS, the File Management System for the disk, usually called DOS (Disk Operating System).

Finally, then, FMS makes a call to SIO (Serial Input Output), the routine which does the actual physical reading and writing to the device. In the case of the disk drive, this involves the actual transfer of a single sector of 128 bytes (or 256 bytes in non-1050 double density).

Most BASIC programmers seldom—if ever—

have need to read or write a physical disk sector. Most avoid writing because it is dangerous, since disturbing the format of portions of a sector can destroy DOS's ability to manage the disk for you. (Reading a sector, though, can be informative, especially if you are trying to either understand DOS or find lost information.)

On the other hand, some programmers like to hide things on a disk. Perhaps high game scores, a password, or some sort of software protection. The best place to hide such information is someplace unknown to DOS.

An Extra Sector

Now, the fact that standard Atari DOS (version 2.0S and its derivatives, including OS/A+ and DOS XL versions 2) leaves sector number 720 available has been documented before: DOS manages sector numbers 0 to 719, but the disk drive understands only sectors 1 to 720. DOS has been fixed to think that sector 0 is always in use, but sector 720 remains outside its ken. Many programs, including some found in old issues of *COMPUTE!*, have read and written data directly to sector 720.

Lo and behold, thanks to a quirk which began who knows how and where, sector 3 is also free for this kind of use! It is the last sector of the traditional three-sector boot process. But for some reason lost in programming legend, it turns out that none of the disk boot code used by DOS is present in sector 3: sectors 1 and 2 contain all the boot that is needed. So, if you are looking for another 128 bytes of hidden disk space, you now know where to find it.

A word of warning, though: If you erase, write, modify, or rename the DOS.SYS file, sector 3 will automatically be rewritten by DOS (it thinks it needs to reestablish the boot code). So, if you choose to use sector 3 for your own purposes, be sure to do so on a disk which either never receives a DOS.SYS file or which has one that you feel is reasonably permanent.

I have not included a program here to access sectors directly because the technique has been shown many times, many places before. For example, *Mapping the Atari* (COMPUTE! Books) gives some helpful hints, and the Atari technical manuals go into SIO calls in some detail. If enough of you write and request a column on this topic, though, I may present more here in the future.

More Hide And Seek

Many game programmers like to hide their signatures in their work, often to the consternation of their employers, the game manufacturers. Famous examples include the message evoked from *Super Breakout* (the Atari home computer version) when you push CONTROL, SHIFT, and I at the same time. Or how about the power dot in the old Atari VCS *Adventure* game, which got you into an otherwise inaccessible room? In fact, the practice is so widespread that some players spend hours looking for these hidden messages in each new game, even the games that don't have any.

Well, it turns out that game programmers are not the only ones who like to get their ego stroked at the same time they put one over on management. Paul Laughton, the prime programmer behind Atari BASIC and Atari DOS (and Apple DOS and . . . but that's another story), told me of one signature that even got into some of Atari's operating system ROMs.

If you want to see this signature, you'll have to find a 1200XL and be patient. Simply remove all cartridges, disconnect all peripherals, and turn it on. Push the HELP key to get to the self-test program, and with SELECT, choose all tests before pushing START. Then wait. The self-test program will cycle through the ROM and RAM tests and the sound register tests before it gets to the keyboard test.

Now how the heck can you have a meaningful keyboard test which is self-running? Answer: You can't. To really test a keyboard, someone should hit at least *some* keys. Nevertheless, the program makes a valiant effort to pretend that it is hitting some random keys. Or does it? Aha! If you look fast and carefully, you will find that the keyboard taps out "Michael Colburn" every single time.

It goes without saying that Michael Colburn had a hand in writing the self-test code. You have to try this on a 1200XL because Atari discovered this signature and changed it when the 600XL and 800XL OS ROMs were produced. The message "Copyright 1983 Atari" is tapped out now, but that's not nearly so interesting as the original.

The Wrong Keyboard

More on the keyboard self-test: It seems kind of sad to me that Atari managed to find the energy and time to change that signature but couldn't see fit to fix the test itself.

If you try the manual mode of the keyboard test on a 600XL or 800XL, you will notice two things wrong: (1) The keyboard layout pictured on the screen is mixed up. The layout shown is

actually the 1200XL scheme, including even the F1 to F4 function keys. (2) The display does *not* show you all legal keypress combinations. In particular, it shows no CONTROL+SHIFT combinations (that is, three-key combinations) at all. And it can't see CONTROL-1 or BREAK. On a 1200XL, the same key combinations are invisible *and* the CONTROL+function key combinations don't display properly.

Well, I always said I thought the self-tests were a waste of valuable ROM space, but it would have been nice if they did their jobs right. (My other objection: If you are going to have self-tests, then test *everything* you can. Like the serial bus, reporting all devices which respond. Like collision detection and other aspects of the GTIA. Like whether the joysticks and paddles work. We have an 800XL which thinks the joystick button is always pushed, but no self-test detects that fact.)

Streamlined Snails

This not-so-little tidbit is a dig at Tom Halfhill. Since he gets to edit my columns before you see them, the very existence of these paragraphs shows his senses of humor and fair play. (See, Tom, I told them you were a nice guy. *Now* will you leave this in?)

In the December 1984 and January 1985 issues of COMPUTE!, Tom wrote a pair of well-balanced and interesting articles on the new MSX computers. If you didn't read them at the time, I urge you to go back and do so. I'm not sure that I agree with all of Tom's conclusions (as you are about to see), but the articles give you the best info I have seen yet on most aspects of this possible new Japanese invasion.

Anyway, the only reason that I bring all this up is that Tom had Assistant Editor Philip Nelson run a simple benchmark program on all the computers that COMPUTE! regularly reports on. Tom then concluded that MSX BASIC showed "streamlined performance." Why'd you go do that? You *know* that I love to eat benchmarks alive. Here goes:

Aside from the fact that the benchmark sorts an array in perhaps the most inefficient way possible, there is nothing wrong with the program as presented. It isn't much good at measuring arithmetic performance, but it is at least as good as the classic BYTE Prime Number benchmark at showing efficiencies (or lack thereof) in logical and branching operations. And the timing numbers presented seem reasonable and correct. So what's my problem?

Well, first of all, I'm a bit tired of seeing little old 8K Atari BASIC pitted against 32K monsters like MSX BASIC. And I don't really like it

when documented, easy-to-use methods of speeding up Atari programs are ignored. Tom says that an Atari 800XL takes 8:55 (minutes:seconds) to run the little benchmark. True. But turn off the screen direct memory access which uses so much CPU time (via POKE 559,0 or with the F2 key on a 1200XL), and the timing immediately drops to 6:10. Which is *faster* than MSX BASIC's 6:20.

Let me play devil's advocate: Isn't this cheating? There are ways to speed up that program on other computers, too. For example, the Commodore 64 loses some time to screen DMA also, doesn't it? Answer: Okay, valid objection. But Atari computers, in general, pay the biggest penalty for text mode screens, and I think benchmark programs should at least include a footnote to this effect or maybe mention *effective* clock rates. (Would it be more legit if we just put a GRAPHICS 19 statement in? That helps almost as much. All right, all right, next subject.)

Faster BASICs

Well, then, how about trying a bigger, more competitive BASIC on that same Atari computer? Glad you asked. BASIC XL handles that program in 4:08. That's two-thirds the time of MSX BASIC and more than a minute and a half faster than the IBM PC. (Just for the record, running the benchmark in FAST mode with the screen turned off gives a time of 2:42, more than twice as fast as the IBM PC. And all these times are *without* the Newell FastChip, which would make even more of a difference.)

I admit I am prejudiced towards BASIC XL. Also, it was handy so I used it first. But another timing of which I am proud is Cromemco 32K Structured BASIC, which handled that program in 4:33 (floating point mode) and 3:13 (integer mode) and which runs on the *same* Z80 processor at the *same* clock rate doing the *same* 14-digit BCD arithmetic as the MSX machine. (And if you count the Cromemco S-10 as a personal computer—which you should if you call an IBM PC or a Commodore 8032 by that name—then I will be glad to dispute Tom's claim that MSX BASIC "may be the most powerful BASIC on any personal computer.")

One more thing before I draw my conclusions: I would very much like to change that benchmark just a little bit. Add lines 1 through 99, each consisting of just a REMark statement, and change the names of the variables to VARIABLE1, VARIABLE2, etc. If MSX BASIC holds true to the standard Microsoft BASIC patterns, its speed will suffer considerably. And so will all the other derived-from-Microsoft times. (Atari BASIC will slow down from the extra lines, but not from the long variable names. BASIC XL in FAST

mode and Cromemco BASIC will not change by even a second.)

So the question becomes: Why is MSX BASIC so doggedly *slow*? Here we have a 32K language running on a very fast 8-bit processor, and it really only shows off halfway decently when you run very small benchmark programs with one- or two-character variable names. Why? Because Microsoft has never significantly improved BASIC. The versions of the BASIC language used in all these machines (even the IBM PC with its so-called 16-bit processor) are still derived from the original Microsoft BASIC designed for an 8K Altair many years ago.

When you are trying to fit a computer language as complex as BASIC into 8K (and that includes Atari BASIC), you *have* to make sacrifices somewhere, and performance is usually the first thing to go. But why, when a computer manufacturer gives you 32K of room for a language, do you need to keep the same scheme? Isn't it time to rethink the methodology of the interpreter? Data General and Hewlett-Packard and Digital Equipment Corporation knew how to build superfast interpreters back before microcomputers were even dreamed of. But they usually ran those languages in 64K memory spaces, the same *total* size as most of today's 8-bit micros.

Optimizing A Language

When we started building Cromemco 32K Structured BASIC *back in 1978*, we had already written an 8K BASIC using many of the same techniques Microsoft used. But since this time we had 32K to work with, we studied the mini-computer languages and started from scratch with better methods. If we were to do it again today, we'd start from scratch *again* and get even more power and better times. Microsoft has never done this.

To be fair, BASIC is not exactly a hot item around Microsoft nowadays. Apparently Microsoft assigns higher priority to other languages, operating systems, operating environments, and word processors, than to redesigning BASIC. Why not leave "improvements" to BASIC to junior programmers, as a maintenance chore?

In his January article, Tom wondered if the hardware technology of the MSX might not be a little tired and boring compared to what other manufacturers will be showing soon. Somehow I can't help but wonder and hope if someday maybe—just maybe—BASIC users will get bored with tired *software* technology, too.

Boy, did I get on my soap box this month. Well, it's relaxing (for me, at least) once in a while, and I promise that next month will bring something different. ©

INSIGHT: Atari

Bill Wilkinson

Atari Acquires Apple!

As I write this, the Winter Consumer Electronics Show (CES) in Las Vegas has just ended. By now you have probably read in the papers and magazines just what real marvels the new Atari Corporation introduced at CES. While I didn't get a chance to attend CES (though others from my company were there), I did have the privilege of getting some preshow information about Atari's new products. Also, thanks to being just a bit nosey, I learned a little about how Atari developed their remarkable new computers and even a little bit of what's yet to come.

Purchase Obvious In Retrospect

(An important aside: The issue of *COMPUTE!* which will carry this article is dated April 1985. However, since this issue will most likely appear on newsstands and in subscribers' mail by about mid-March, you might be reading this before April. If so, be sure to keep all of what I am about to reveal secret until at least the first of April.)

Reveals Other Buys

Anyway, as I started to say, I was lucky enough to be privy to some early information and (thanks to my nosey nature) overhear even more. One thing I overheard was a simple question, "Should we take the Mac with us?" (An obvious reference to an Apple Macintosh.) It seems that in the process of designing the 130ST and 520ST computers, the engineers at Atari looked at several existing computers. Now, no rival companies were about to be so generous as to donate machines. So, looking back, it seems obvious that Atari had to go out and buy several—including the Mac, of course.

IBM Failure Described

In the process of evaluating the various computers, Atari also was able to look at the microprocessors (CPUs) which they used. It comes as no surprise that the 8/16 bit 8088 used by the IBM PC was rejected early on as being unable to achieve the speed Atari desired. So what processor got the nod for the 130ST and 520ST?

Leonard Tramiel Departs Company

Although I have managed to enjoy Leonard Tramiel's company in several meetings, the one time we managed to get in a really interesting discussion of processors he had to depart early (for another meeting, probably). Before he left, he did seem to indicate that his personal choice for a CPU might be the National Semiconductor 32016 and 32032 processors. They are very powerful and very orthogonal machines, but (and this is speculation on my part) the fact that they are available only from National Semi probably makes choosing them difficult for any company.

In any case, Atari chose to go with the tried and true Motorola 68000 series of processors, the same one used in the Apple Macintosh and Lisa computers. (An aside: The official meaning of the ST designation is "Sixteen/Thirty-two" for the 16-bit bus and 32-bit registers of the 68000 chip. XE implies XL compatibility, but Extended.)

Future Plans Fall Flat

What about all the loyal Atari 400/800/1200XL/600XL/800XL owners? Has Atari completely forgotten them? *No way!* Apple has Mac and Lisa, both built around a 68000 chip, in its "sort of 32-bit" division, and the IIe and IIc, both using a 650x CPU, in its 8-bit division.

Lo and behold! We already saw that Atari

has the 130ST and 520ST built around the 68000. Does it really surprise you to learn that the 65XE and 130XE will be produced using a 650x processor? And we were even given the privilege of having a set of drawings for a portable computer (in the 650x line) dropped flat on the table in front of us!

Original Projections Unrealized

The same day we saw those plans for the portable, we also got to see some of the features that the new machines will be sporting. On that day I decided that my predictions of success for Atari, which I made in this column in December, could very well have been ridiculous underestimates.

Operations Shut Down

What kind of features impressed me? I think it will be obvious to you when you read a spec sheet at your local dealer or the other CES coverage in this issue. In the meantime, I'll give a brief list of what I think are the best features of each machine at the end of this column. I tried to ask some of my contacts at Atari about a couple of things I am not quite clear on, but the lure of CES left the software and engineering departments virtually shut down for these four days.

Long-Term Outlook Bright

If there is any area of concern to those of us here at Optimized Systems Software, it is about those products where our software sales overlap those of Atari Corporation. New prices on Atari software have made us rethink some of our plans, but we think that there will always be sophisticated and/or advanced users out there who will be willing to pay a little more for higher quality. And we are not alone: The number of companies showing Atari-compatible software or hardware at CES was almost amazing. Will we stay in the Atari software market? How could we not?

At Last

"What the heck," you ask, "was all that about?" The answer: Every word that you just read was true. Even the subheadlines are properly explained in the text. Oh, I may have bent some words here and there to make the headlines more spectacular, but that was the whole purpose of this exercise. I always wanted to show how you can take an innocuous and/or positive review and generate sensational *National Enquirer*-type headlines.

If you're an acrostics fan, you may have already caught the significance of the first letter of each headline. (Go back and reread them if you want a minor laugh.) This is, of course, my annual attempt at some humor. It's not very subtle or well-hidden this year, because I thought it

would be fun to find out how many COMPUTE! readers actually plow through all my verbiage. If you got to here unscathed, congratulations. Time for a complete change of pace.

New Machine Features

This is just a simple table of what I feel are the most important features of four of the new Atari machines. I am sure that more info will be available by the time you read this, but maybe these specs will whet your appetite.

65XE

- 6502-series processor.
- 64K of RAM.
- Very, very compatible with 800XL.
- Nicely sculptured case and keyboard.
- Cartridge port on rear (where our ugly orange cartridges won't be so obtrusive).
- About \$100.

130XE

- Identical to 65XE *plus*:
- 128K of RAM (supported as a ramdisk by new DOS 2.5).
- Expansion port on rear (used in conjunction with cartridge slot).
- About \$150.

130ST

- 68000-series processor.
- 128K of RAM.
- 192K of ROM.
- Uses Digital Research's GEM windowing and display system—virtually identical in form and function to Apple's Macintosh system.
- Built-in RS-232 interface.
- Built-in parallel printer interface.
- Built-in disk controller handles up to four floppy disk drives (designed to use *very* inexpensive 3.5-inch drives, 360K each—priced perhaps as low as \$100!).
- DMA-capable expansion port (designed for *very* fast hard disk drives).
- Three-voice sound chip.
- Color graphics (640 × 400 in black and white, 640 × 200 with four colors, 320 × 200 with 16 colors).
- Cartridge slot (up to 128K ROM in cartridge).
- 10 special function keys.
- MIDI interface (for music synthesizers and ???).
- About \$400.

520ST

- All the features of the 130ST *plus*:
- 512K of RAM instead of 128K.
- About \$600 (Yep . . . that gives you a color "Fat Mac" at around \$1,000).

Information Please

It's time, once again, to respond to some letters. I may have made a mistake in publishing the P.O. box where you can write me directly, since I find myself with about five or six times as much mail to answer as I had before. Until I get adjusted to answering this much correspondence, please bear

with me.

For this month, I have decided to select some letters which (I think) really *need* answers. Surprisingly, for such varied topics, the answers to all may be much the same.

Bob Dorn, of College Park, Georgia, was the first of three or four to ask me how to use an Atari 1030 direct-connect modem to upload and download files. Well, you got caught in the great Atari let's-protect-the-poor-dumb-user game. For reasons best understood only by now-extinct marketing people at the old Atari, neither the 835 or 1030 modem came with software support for uploading and downloading programs, text files, and so on. I guess those marketers never used a computer with a modem, so they couldn't see any use for the capabilities.

Luckily, many other people, including a few software gurus, found themselves in the same fix you are in. One commercial company which seems to be doing a lot of work with these modems is Gardner Computing, P.O. Box 388, Holbrook, NY 11741. I am *not* endorsing them (I have never used any of their products—I have only read their ads), and I apologize in advance for inadvertently slighting any other companies supplying similar software.

There are other solutions. See the "Readers' Feedback" letter headlined "Atari Modem Update" in the February 1985 issue of *COMPUTE!*. There are also some programs floating around in public domain user group libraries which allow upload/download and more. As a general rule, such programs come without documentation (or, at most, with a few paragraphs on the disk with the program), so you may need to do a little detective work to use them.

Good Local Support

Again, though, there may be another solution. *Join your local user group.* Come on now, what will it cost you? One evening and a couple of dollars a month will probably be the best investment you ever made in computing. And so many user groups have people who know the answers. To almost anything you ask!

Another practical reason for joining such a group is that Atari has already announced that its primary means of providing programming support to users will be through the user group network. The toll-free phone lines are gone, and the support group is decimated. This may be the *only* way to get technical answers in the future (aside from writing to me or "Readers' Feedback").

All of this, and we haven't even mentioned the fact that most user groups have literally *hundreds* of programs available for next to nothing. Okay, okay. Some of the programs don't work

right, are poorly written, are too slow, etc. So what? You are getting what you paid for and more. If nothing else, a cruddy little Atari BASIC subroutine may lead your computer to uses you hadn't thought of yet.

So join, join, join. Why wait five months for my answer to appear in this magazine when help is available two miles from your home?

How do you know where/who/when/what your local group is? Well, try asking at local computer stores, even those that don't sell Atari products. Look in your local paper. Look in Atari-oriented magazines, which sometimes have listings of clubs. If you are really desperate, send me a *self-addressed and stamped* card or envelope. No guarantees, because I don't know where *all* the clubs are, but if there's one on my list I will tell you. *Please* use me only if all else fails, because (1) I'm always too busy, (2) it may take me some time to answer, and (3) if I ask my kids to help me with this, they will charge me.

Deluged With Information

From going to users who can't find what they need, we go to a couple of readers who have found too much. Jamie Patterson, of Hooker, Oklahoma, sent me a well-argued plea for some help in choosing material about his three-month-old baby, an 800XL computer. I quote: "How does a three-month-old know which books to choose?"

Darned good question. My usual answer, when I want to choose a new computer book, is to go to two or three bookstores that carry a couple of hundred computer books each and browse. This works because there are at least a dozen such bookstores within reasonable distance of my house. Now, I have to admit I don't know where Hooker, Oklahoma, is, but if it isn't within 20 miles of a major computer bookstore, my method won't work for Jamie. What can he do?

The editors of *COMPUTE!* might like me to answer, "Buy a *COMPUTE!* book." But whatever book you buy, you must choose one which is at the right level for you. From *COMPUTE!* Books, the most general material may be found in the *First*, *Second*, and *Third Book of Atari*, along with the two books on *Atari Graphics*. Some, but not all, of this material is relevant to someone who has learned the fundamentals of Atari BASIC.

Suppose, though, that you aren't even to that level yet. You don't know a *PRINT* from a *PLOT* statement. Where do you turn? Since Atari stopped shipping copies of *Inside Atari BASIC* with the XL computers, buyers have been left to choose their own tutorial. And what should they choose?

My trouble is that every time I look at a book that purports to teach BASIC (or word

processing or assembly language or . . .), I find something wrong. I don't like the order of presentation of the topics. There are mistakes in the section on how to speed up your programs. The author encourages poor programming style. The list goes on and on. So I refuse to make a firm recommendation.

The Great Book Survey

What, then, can Jamie Patterson and others like him do? What else? Join a user group. Ask other Atari owners. Ask to look at their books. Okay, so maybe none of the over-200 user groups is close enough to Jamie. And, besides, he asked *me* for an answer. I guess I should do something, right?

So here it comes. I am asking you, my readers, to make some comments on the books *you* have learned from. Don't stick to learning BASIC. Any aspect of Atari computers is eligible, even manufacturers' manuals. To make life easier for me, just send the title(s) of the book(s), the level (1 to 10, with 1 being rank beginner), and your overall rating (0 for trash to 10 for perfection). A postcard will do fine.

I don't want any experts evaluating these books; I can mishandle that aspect myself. Instead, I want actual real-life experiences. Did or did not the book teach you what it said it would? If it did, was it an uphill battle or did the style make it downright easy for you? I can't respond personally to these rating cards, but I will report the results received by April 20 in the August or September issue (sorry, but that's the fastest turnaround possible).

Translators, Again

Robert Glover, of Cleveland, Tennessee, has been the proud owner of an Atari 400, an 800, and now an 800XL. He asks me why he can't simply use the binary save option of Atari DOS to make a copy of the 800's operating system ROMs and then load that file into his 800XL as a home-brew translator disk. He suggests that I perform this service in my column.

Well, in theory, and with some modifications to his method, I *might* be able to do so. Why won't I? First, there are several problems to overcome. Two of the simpler examples: (1) You can't write/save ROM directly with DOS 2.0S; you have to copy it down to RAM first. (2) Joystick ports 3 and 4 are used for *output* in an 800XL and for *input* in an 800.

Also, how many readers have access to both an 800 and 800XL? And, finally, why go to that kind of trouble when the translator disks are so available?

Ah, but that last point was raised by Mr. Glover. He says he cannot find the translator

disks anywhere. Hmmmm. Guess where I am going to suggest he look? Right. Ask your local user group. And that brings us back to the quandary of the last reader: What if there is no user group nearby?

I have a couple of partial solutions. First, there are a few mail-order organizations which, in addition to selling commercial software, sell public domain programs for reasonably low prices. Right now, LotsaBytes (15445 Ventura Blvd., Suite 10, Sherman Oaks, CA 91413) seems to be the leader in this category, but I should also mention DynaComp, Antic, and ANALOG (the latter two offer primarily games and BASIC utilities from their magazines).

Perhaps even better, many user groups (especially the larger ones) allow mail-order memberships. Since there are so many of these groups just crying for members, I hesitate to recommend one over another. But because their newsletter has been around the longest and may have the greatest number of readers, I will at least mention the very friendly people of ACE (3662 Vine Maple Dr., Eugene, OR 97405).

So my message this month is clear: Atari is very, very, very much alive and well. Keep your interest in your machine similarly healthy by joining a user group.

C

WOW!



IBM PC w/Drive \$1339.95
APPLE 2E w/DRIVE \$809.95

"PRINTER SPECIALS"

Anadex 1034	Epson FX 100+ 564	Okimate 10 130
Brother HR15 XL 349	Epson LO 1500 1029	Olympia 10 314
Brother HR 25 564	Gemini 10X 227	Panasonic KXP 1091 259
Brother HR 35 799	Gemini 15X 329	Panasonic KXP 1090 174
Brother Keyboard 129	HP Laser Jet 2695	Panasonic KXP 1092 387
Citizen MSP 10 329	Juki 6100 Televideo 367	Panasonic KXP 1083 367
Citizen MSP 15 488	Juki 6300 749	Panasonic KXP 3151 450
Corona Laser 2409	Mannesman Spirit 80 199	PowerType 289
Daisywriter 759	Mannesman 180L 544	Quadjet 720
Delta 10 324	Mannesman 180L 742	Radix 10 481
Delta 15 444	NEC 2050 659	Radix 15 567
Diablo 620 API 699	NEC 3550 1325	Riteman Blue + 279
Dynar DX 15 XL 349	NEC 7730 1859	Silver Reed Exp 550 378
Epson RX 80 FT + 294	NEC 8650 1709	Silver Reed Exp 500 286
Epson RX 80 226	Okidata 92 349	Silver Reed Exp 770 715
Epson RX 100 394	Okidata 93 564	Toshiba 1340 678
Epson FX 80 384	Okidata 84 660	Toshiba 1351 1206

ZENITH

PC2 150 1649	8201A 8201
PC151-52 2095	8801 8801
Z161-52 2244	

NEC

308 800 XL 108
699 1027 229
1050 Drive 154
Indus Drive 279

ATARI

108 800 XL 108
229 1027 229
1050 Drive 154
Indus Drive 279

IBM

PC w/Drive 1339
PC XT w/Drive 3074
Monitor Card 159
Color Card 169
IBM Monitor (GRN) 194
Hercules Mono Card 299
Hercules Color Card 154
Tecmar Captain 64K 269
AST Six Pack 84K 259
Taipei 20 Meg 2395
Piantronics 384
Keytronics 154
IBM Drive 239
Paradise Multi 292
4 M Drive from 99
10 Meg Drive 849
Bernoulli 2029

COMMODORE

Commodore 64 152
1541 Disk Drive 184
1702 Monitor 189
MPS801 Printer 179
1526 Printer 194
1650 Modem 89

MODEMS

Hayes 1200 439
Hayes 1200B 378
Hayes 300 184
Micromodem 2E 214
Access 123 364
Novation J-Cat 89

SANYO

550 S.S. 847
550 D.S. 659
555 S.S. 829
550 D.S. 974
CRT 30 99

APPLE

2E w/Disk Drive 809
Macintosh 1679
Apple 2C 892
ImageWriter 486
AppleScript 269
Addl. Drives From 1200 Modem 444

MONITORS

Amdex 300 Green 114
Amdex 300 Amber 124
Color 300 324
Color 600 384
Color 700 489
310 Amber 140
Taxan 210 205
Princeton HX12 459

For Your Protection We Check For Stolen Credit Cards
 Some Items Reflect Cash Discount

HARMONY VIDEO & COMPUTERS
 2357 CONEY ISLAND AVE. BROOKLYN, NY 11223
 800 VIDEO84 OR 800-441-1144 OR 718-627-1000

INSIGHT: Atari

Bill Wilkinson

More About HELP! On HELP?

Several of you were kind enough to write and point out (with only a few snickers) that I goofed in my February column's description of the HELP key. Specifically, I gave you the wrong value for SHIFT+HELP. Here is the corrected table. Remember, though, that you must POKE location 732 back to zero if you PEEK there and find that the HELP key has been pressed.

Key(s) Pressed	Value in 732 (\$2DC)
HELP alone	17 (\$11)
CONTROL+HELP	145 (\$91)
SHIFT+HELP	81 (\$51)

B Is For Bad BASIC

I was inundated with letters from people who responded to my request for help in that same February issue. I had asked if anyone knew how and why the Atari BASIC built into the XL machines caused the infamous keyboard lockup. As I stated then, I was under the impression that the oh-so-little (but oh-so-damaging) coding mistake which caused the problem with Atari cartridge BASIC had been fixed.

Well, it turns out that I was both right and wrong. I was right about that particular bug being fixed. I was right in believing that Atari had a version of BASIC which corrected the problem. What I had not been aware of was the number of 600XLs and 800XLs that Atari has sold which contain an intermediate version of BASIC with even more severe problems.

If we call the original Atari BASIC *revision A*, then the most current version being shipped and installed by Atari (in XE machines as well) is *revision C*. So what about *revision B*? In fact, Atari gave me an early release of rev B in cartridge form. ("Rev" is the usual contraction of "revision" if you're into techie language.) However, when I learned that it had significant problems and that Atari was dropping it in favor of rev C, I promptly ignored and forgot about rev B.

Unfortunately, Atari didn't do likewise. Atari (the old Atari, that is) ordered a few thousand (tens of thousands? hundreds of thousands?) ROMs using rev B, which they certainly weren't going to throw away, so kerplunk into all the 800XL and 600XL computers they went.

As I said, I had kind of ignored rev B be-

cause I was under the mistaken impression that very few machines using it had been shipped. Boy, did my mail tell me I was wrong! So now, how can I help all of you out there who are stuck with rev B BASIC? Three ways: First, show you how to tell what revision of BASIC you really have. Second, tell you how to avoid the problem most of the time. Third, tell you how to fix the problem permanently.

What Have I Got?

I am indebted to Matt Ratcliff for showing me a location within Atari BASIC which tells you what version of BASIC you have.

If you
PRINT PEEK (43234)
and see this value:

Then you have
this revision
of Atari BASIC:

162
96
234

A
B
C

Despite what you may have heard or read from other sources, there is *no* practical way to avoid some of the problems associated with rev B. Many Atari "experts" won't believe me, but that's not surprising. Even though we wrote—and, in 1983, COMPUTE! published—*The Atari BASIC Source Book*, with the complete source code of Atari BASIC rev A and a detailed explanation of the keyboard lockup bug, I saw a user group newsletter just three weeks ago in which someone claimed that hitting SYSTEM RESET cleared up the problem. Honest, there is *no* reasonable way to avoid the bug in rev A, either.

However, there is a way to minimize the effects of the worst bug in rev B: Don't use the SAVE or CSAVE commands. Instead, use LIST and ENTER. (Disk users simply substitute the words LIST and ENTER for SAVE and LOAD, respectively. Cassette users use LIST "C" and ENTER "C" in place of CSAVE and CLOAD.) Even this technique will *not* help you avoid the bug. It will just make it easier for you to recover when you get bitten.

In a nutshell, the problem in rev B is that your program and/or your data can get hopelessly scrambled. Unlike rev A, though, you may not notice the scrambling until some time after it first occurs, since the scrambling often does not cause a lockup. How can you tell if your data is scrambled? You can't, easily. How can you tell if

your program is scrambled? Just LIST it on the screen or a printer. If it looks okay, it probably is okay.

So start by deciding how much time you are willing to throw away, if worst comes to worst. (For me, that's about 15 minutes. If I were using cassettes, I might make that 30 minutes.) Then, every time you have typed that much time's worth of new material into your program, LIST the program on the screen or printer to be sure it's okay. If it is not okay, even if some lines just look funny or scrambled, *turn off your power and reboot*. Do not attempt to fix your program. The odds are you will only make the situation worse. Only after rebooting should you re-ENTER the last listing from your disk or cassette.

If the screen or printer listing appears okay, you can go ahead and LIST the program to disk or cassette. This way you can have reasonable confidence in that version if you need to re-ENTER it later.

Alternate Solutions

Sidelight for all Atari cassette users: The technique I just described is a good idea no matter what version of BASIC you are using. Remember that you can easily verify a LISTed tape by re-ENTERing it back over itself. Do *not* type NEW before using ENTER "C". If a tape has an error, the most you will wipe out using this trick is one line. If it has no errors, the ENTER will terminate normally. (Disk users may also use this verification trick, but it seems unnecessary if you always use write-with-verify mode on the disk. Atari DOS defaults to this mode.)

You probably noticed that I said there was no easy way to tell if your data (strings, arrays, etc.) had been scrambled. As far as I can tell, though, any scrambling in these areas is fixed every time you use the RUN command. (If you want to feel super-safe, type NEW and re-ENTER your last LISTed version.) And there appear to be only two ways the bug can occur while a program is running: (1) If you ENTER an overlay in the middle of your program. (2) If you DIMension a string or array when you are several levels deep in a GOSUB and/or FOR nest (several means 64 GOSUBs or 22 FORs or some combination of the two which uses about 256 bytes of stack space).

Maybe the best solution of all is to forget about rev B BASIC entirely and get a different BASIC for your computer. You could buy one of the enhanced BASICs available on disk or cartridge from several independent companies. Or you could buy one of the new XE-series computers, which have rev C BASIC built-in. Or you can order a rev C BASIC cartridge from Atari it-

self. Perhaps to atone for the bugs in rev B, Atari is offering the rev C cartridges at a nominal cost. Send \$15 (no extra shipping and handling charges) to:

Atari Corp.
Customer Relations
390 Caribbean Drive
Sunnyvale, CA 94088

The cartridge works with all eight-bit Atari computers. (Remember that when you plug in a cartridge on an XL or XE, the built-in BASIC is disabled and control passes to the cartridge.)

Bits And Pieces

When I told you above that you need to LIST your program periodically, did you automatically start allocating two or three cassettes or a blank disk for each program? If you didn't, you might as well ignore my advice. *Never* use the same cassette or same filename on a disk to keep successive LISTings of SAVED programs. If you have a good version of a program SAVED as "D:MYPROG.SAV" and then, in the process of adding more lines to that program, you encounter one of the nasty editing bugs, what happens when you SAVE it again with that same filename? You just wiped out your last good copy.

At the very least, keep the last *two* versions of every program, every word processing document, every data file, etc. (I always keep three copies and usually keep at least as many as a blank disk will hold.) Unless, of course, you value your own time at less than 25 cents an hour.

More than a few of you have written with questions about my enhanced DOS 2.0S (from COMPUTE!, August and September 1984) for the 1050's dual density. First, I want to thank you for the nice words and shrug off the complaints. Then, I have the pleasure of telling you that Atari will soon be releasing DOS 2.5, which uses 1024 sectors (out of a possible 1040) on a 1050 drive in dual density mode. It is very, very compatible with DOS 2.0S. Do I have to tell you that it's completely incompatible with my earlier version? I don't? Good, then I won't mention who helped write it for Atari.

The ink wasn't dry on the March issue of COMPUTE! when I started getting letters (and even two phone calls) asking me to please explain how to read/write a sector directly from/to the disk. I said I would oblige if enough of you asked, so sector I/O, and a recap of Atari I/O in general, is a future topic. But first, next month's column will explain the bugs in rev C BASIC in more detail and even divulge *how* they happened.

INSIGHT: Atari

Bill Wilkinson

Analyzing The BASIC Bug

Last month I showed some ways to minimize the problems caused by the bugs in revision B Atari BASIC (the built-in BASIC in the 600XL and 800XL). But many of you are curious about exactly *why* these bugs happen, and what effect they can have on your programs.

Let me begin by telling what did *not* cause the error. Rev B BASIC has a peculiar problem: Each time you LOAD (or CLOAD or RUN "filename") a program, rev B adds 16 bytes to the size of your program. If you then save the program, the next time you load it in it grows by another 16 bytes, and so on.

Now believe it or else, these additional 16 bytes were put in deliberately. It seems that there is a minor, undocumented bug in the Atari S: (graphics screen) driver. Under some circumstances, it will use a few bytes below MEMHI (contents of locations \$2E5-\$2E6, 741-742 decimal). So, if you have a program which extends right to the very top of memory, you can wipe out a little bit of the runtime stack where GOSUBs and FORs are remembered. Somebody at the old Atari apparently had the bright idea that if BASIC told you that memory was full when your program got within 16 bytes of MEMHI, the screen/BASIC conflict could be avoided.

A Fix Gone Sour

Pretty good idea. Except for a few problems. First, BASIC doesn't save the string/array space of the runtime stack; both are created when a program is run. So the nice fact that the saved file is guaranteed to have 16 bytes of space left is negated as soon as you DIMension a string or an array or use a GOSUB or FOR. Second, the 16 bytes are added to all of BASIC's size pointers before the comparison with MEMHI is made. Good. But the newly increased value is then stored as the new pointer value. That effectively moves the

program upward in memory by 16 bytes, meaning that the desired 16 bytes of free space aren't there anyway!

Well, the point of this digression is twofold: (1) This is yet another reason to use LIST and ENTER with rev B BASIC, since ENTERing a program does *not* trigger this silly 16-byte bug. (2) Several people wrote and suggested that this 16-byte bug is what causes the infamous keyboard lockup bug. Sorry, folks.

Last month, I mentioned the detailed explanation of the rev A Atari BASIC lockup bug which is to be found in COMPUTE!'s *Atari BASIC Source Book*. Well, apparently somebody at Atari read the book. Or maybe they just noticed that my company had fixed the lockup bug in one of the three or four revised versions of BASIC that we did for Atari back in 1979 (yes, that's 1979). It turns out that the lockup resulted from two missing instructions (and a total of two bytes) within the routine which "contracts" memory. (We say "contracts" because it is used when you delete a program line, so the program is contracted in size.)

Then that same somebody looked at the "expand" routine and saw almost identical code. "Aha!" they say, "Methinks there is a bug here which just hasn't been discovered yet!"

If It Ain't Broken . . .

But they were wrong. The reason the bug appeared in the contract routine is because that routine was written *after* the expand routine and copied its pattern too closely. So our unknown someone at Atari blindly added code to the healthy expand routine and introduced a very nasty new bug. In fact, because this bug appears when you add lines to an existing program, it is probably more likely to occur than the original rev A bug!

To see a demonstration of the bug, enter the following statements in direct mode (without line numbers):

```
DIM A$(249)
A$="ANY STRING YOU LIKE"
PRINT A$
PRINT A$,
PRINT A$,
```

The last two statements won't print A\$ properly in either rev A or rev B Atari BASIC—in fact, they'll mess it up two different ways. Cute, eh? The problem is that tacking that comma on the end of the PRINT statement moves the string/array space (and thus A\$) by one byte. Except it doesn't, really, so the variable value table address of A\$ points to the wrong place in memory! Imagine your program being destroyed in a similar way. Is it any wonder you experience keyboard lockup and scrambled listings?

What former Atari employee did I bribe to get all this information about the bugs in rev B BASIC? Did I get the listing on a microdot hidden in a pack of blank disks? Sorry to disappoint you, but I did what any other hacker would do: I dragged out my trusty machine language debugger and carefully disassembled certain portions of rev B BASIC.

Finally, here's how the two bugs we have discussed were fixed in rev C BASIC, which is built into the new XE series computers (and also is available for older Ataris on cartridge at nominal cost—see last month's column). Since both bugs were caused by adding things to code which worked before, you would think that Atari could simply take the "fixes" back out. Nope. Instead, they patched over the extraneous instructions with what are effectively NOP (NO oPeration) instructions. Tacky? Well, I've certainly done it to DOS here in this column enough times, so who am I to say?



INSIGHT: Atari

Bill Wilkinson

Bargain Basement Networking

From time to time a new product comes to my attention which stands above the rest in terms of performance and value. A recent example is the MicroNet from Micro Peripheral Products (MPP) of Albany, Oregon. The MicroNet is a wondrously simple device that allows you to connect up to *eight* Atari computers (though not the new ST machines, yet) to a single printer and one or more disk drives. You simply connect a standard Atari serial cable (the kind that goes from the computer to the disk drive, printer interface, etc.) from each computer to any of eight sockets on the deceptively small MicroNet box. Then you connect a similar cable from the MicroNet to the drives and printer, just as if the MicroNet were an Atari computer. The result? All eight computers think the disk drive(s) and printer are their very own! Well . . . almost.

This is *not* a sophisticated high-speed network with several megabytes of mass storage and an automatic printer spooler online. It's still using the clunky 19,200-baud Atari serial bus, slow enough when only one computer is using a drive. With eight computers, you may have to wait eight times as long to read something from a disk (though a delay this drastic is unlikely). And what about printing? You sure don't want to use the slow Atari 1027 printer in this configuration! Still, let's take a look at situations where this system makes sense.

First and most obvious is the classroom. A teacher can put the day's lessons on a disk from any one

of the computers, write-protect the disk, and then let each student boot his/her own computer and start using the appropriate materials. Or the teacher can boot each computer—it would take only two or three minutes. Reports on each student's performance could be kept on a second disk or printed on the shared printer.

The total cost of this system, assuming eight computers? Look at the chart below. Prices are rounded up from retail, and an enterprising dealer should be able to offer a substantial discount on a package like this:

8 800XL or 65XE computers	\$ 800
8 color TVs or monitors	\$1600
2 disk drives	\$ 400
1 fast printer	\$ 400
1 printer buffer	\$ 200
1 MicroNet	\$ 200

TOTAL \$3600

Cheaper Than Terminals

Surprised by the last two items? The printer buffer is recommended by MPP. By spooling printer data into the buffer at high speed, a single computer won't tie up the MicroNet bus for so long. And if you were surprised at the low cost of the MicroNet, you read it correctly: The actual suggested retail price is only \$199.95. Hard to believe!

That puts the per-station cost at \$450, less than the price of a black and white, nongraphics *terminal* on a conventional time-sharing system. Or about one-third the cost per station of an Apple IIc network system.

Could the MicroNet be used for business applications? Well, maybe. A big fat maybe. The MicroNet provides *no* file protection whatsoever. No password security. No way of

stopping user No. 2 from zapping user No. 1's files. Etcetera. And there's certainly nothing to prevent two users from trying to write to the same disk file at the same time. Lots of potential problems. The easiest solution is to write software which is alert to the possible problems.

For example, the MicroNet gives exclusive control of the disk drive to a single computer long enough for it to create a file. A program running on another computer could look for the existence of that file as a signal that it could not write to a certain database file. Sounds clumsy, but many of the cheapie time-sharing systems of the 1960s and 1970s had this problem and solved it the same way.

The MicroNet system can definitely be crashed if its users are hostile, and that's one reason I suggested that teachers write-protect their master disks before letting all the clever kiddies take a crack at crashing it. (My own little seven-year-old knows that crashing a disk means he doesn't get to play on the computer for a while. He is now beautifully conscientious about popping the disk before turning off the power.)

The MicroNet is obviously an economical solution to some problems. It is not all things to all people; but, at its price, it is certainly worth looking at. (For more information, write to MPP at 225 Third Avenue SW, Albany, OR 97321.)

Next month: Part 1 of my long-promised series on Atari input/output. Theory and a little bit of practice. See you then. **C**



Atari Input/Output

Much of what I'm about to discuss this month has appeared in this column before. And the bulk of this information can also be found in the *Atari Technical Reference Manual*—presuming you can read "techlish." But this intro is necessary so we can start talking about the meat of our subject next month.

Still with me? Let's go. Atari's operating system (OS)—which, like the OS in any eight-bit computer, takes up the bulk of Read Only Memory (ROM)—is really a thing of beauty. In fact, it may be the only *consistent* OS to be found in any microcomputer, short of those sporting UNIX or its derivatives. CP/M and MS-DOS are such kludges that most commercial programs bypass the OS. (That's why there are so many "almost PC-compatibles.") The Commodore 64's operating system comes close, but its disk input/output is difficult at best. And Apple's ProDOS manual states that "users desiring to perform I/O to devices other than the disk drive are on their own!"

Atari users, on the other hand, enjoy a system with such complete support that, for most programs, all necessary input/output operations can be executed by calling a single subroutine! That subroutine is called, appropriately, *Central Input/Output* (CIO). By calling CIO with the proper values in certain memory locations and the proper pointer in the 6502's X register, your programs can perform such diverse operations as formatting a disk, drawing a line on the graphics screen, fetching a keystroke from the keyboard, sending output to the printer, or reading 25,000 bytes from a disk file.

Yet, CIO is invisible to most Atari users. For example, many of the capabilities which magazine and newsletter articles attribute to

BASIC are not part of BASIC at all. None of the graphics (including the so-called BASIC graphics modes) in Atari BASIC are actually performed by BASIC. Instead, BASIC simply translates the graphics command into a call to CIO. Atari developed this system in 1978, and it wasn't until the Macintosh appeared that such a revolutionary concept was repeated in a popular computer.

Generally, you have to become a machine language programmer to appreciate and use all the features of CIO. So why read any of this, then? Because calls to CIO can't perform *every* input/output task possible on Atari computers. What can't CIO do? First, let's take a glance at what it can do.

Calling CIO

When CIO is called by a program, it expects the X register to contain a pointer to an *Input/Output Control Block* (IOCB). IOCBs are blocks of memory 16 bytes long which control CIO functions. The pointer value for the X register is easily calculated—it's actually the BASIC file number (as in OPEN #1,4,0,"K:") multiplied times 16, because there are 16 bytes per IOCB. One of the bytes within the IOCB then tells CIO what function the program is requesting.

There are seven fundamental functions available: OPEN, CLOSE, STATUS, PRINT, INPUT, Block PUT, and Block GET. In addition, there are some *extended functions*. BASIC programmers are familiar with these because of the XIO statement, which allows you to call the functions from BASIC. But several other BASIC statements (including NOTE, POINT, DRAWTO, and LOCATE) access the CIO extended functions, too.

After CIO examines the IOCB and determines which function is being requested, it decides which

device (keyboard, disk, screen, etc.) should service the request. Then it calls an appropriate routine within the *device driver* for that device. (For example, the Disk Operating System—or more properly, the File Management System—is the device driver for the disk drive.) If the request is for an extended function, it is passed on unchanged to the device driver.

Well, with 256 possible command values, you would think that there isn't any request, however bizarre, which couldn't be serviced via CIO. In theory, true. In reality, you have to stop adding functions somewhere or you run out of memory. Thus Atari's CIO-based graphics have no function for drawing a circle, and DOS provides no command to format a disk without also writing a boot and directory.

If you want to draw a circle, you can write a routine to calculate and PLOT points or change screen memory directly. If you want to mess with the disk drive, though, you have to learn about another routine within the Atari ROMs, *Serial Input/Output*.

The Mysterious SIO

SIO—which lets Atari computers talk to devices (such as printers and disk drives) which hook up to the serial bus—has acquired an underserved aura of mystery. Actually, though, in some ways it is easier to call SIO than it is to call CIO!

For example, there is only one SIO "device" and only one Device Control Block (DCB). So even the X register pointer required by CIO isn't necessary when calling SIO. Intrigued? I hope so, because it's time to sign off for now. But after this brief overview, we're ready for next month's column: We'll show how to write a program to call SIO.





Using Serial Input/Output

Last month, I introduced the structure of Atari's operating system (OS). My most important point was that the OS consists of several layers. When you type in a BASIC statement such as LPRINT "Hi There!", you cause a fairly complex chain of events. First, BASIC figures out that LPRINT means you want to use a printer, so it calls the OS to open a channel to the printer (always channel number 7, in this case). Then BASIC sends the bytes to be printed to a part of the OS called Central Input/Output (CIO), which in turn realizes that a file to the printer has been opened on that channel. CIO calls the printer driver, which collects bytes until it has a block of them (or until it gets a carriage-return character or a CLOSE command). Finally, the printer driver sends a block of bytes to the printer by calling *Serial Input/Output* (SIO)—another subroutine inside the OS, and the subject of this month's discussion.

I'd like to point out that this process stops at SIO only as far as the computer is concerned. The printer interface (for example, an 850 Interface Module) also contains a microprocessor which collects the block sent to it by SIO. Then the interface passes the block, a byte at a time, to the printer. Within the printer, yet another microprocessor is usually employed to control the various motors and hammers and wheels that actually place the characters on paper.

Did you note that the process of printing even a single character most probably requires the use of three microprocessors? Did you stop to think that each of these processors requires software to make it work? Did you ever wonder why there are so many people making a living at programming? (Though barely, in the case of some of us.)

Perhaps the most amazing thing is that, for the most part, the

three microprocessors work reliably and efficiently together. (It is even more amazing when you consider that either the printer or interface module is often made by a company other than the one which made the computer!) The secret to success here is standardization. The usual printer connection is a fairly simple one, originally defined by a company named Centronics and now adopted by almost every manufacturer in the microcomputer market.

The way your Atari computer "talks" to your interface module, though, is strictly an Atari invention—the SIO. There is a well-defined protocol associated with SIO. It includes such niceties as Command and Data Frames, Acknowledgment, Nonacknowledgment, Command and Bus Errors, and more. Luckily, 99 percent of all Atari programmers need never learn these gory details, since there really isn't anything you can do to change their workings.

Disk Access Via SIO

Some programmers, however, do want to send and receive blocks via SIO. And usually the blocks to be transferred are disk sectors. So let's look at how one reads or writes a specific disk sector.

When SIO is called by a program, it expects to find certain information in a *Device Control Block* (DCB). There is only one DCB, located at \$0300-\$030B (768-779 decimal). It contains four one-byte values and four two-byte (word) values, all of which must be set up properly. The accompanying table briefly describes each location in the DCB. See *COMPUTE! Books' Mapping the Atari* for more details.

Does all this look confusing? Not to worry. Program 1 below is a subroutine which does most of the work for you. Just type it in, LIST it to disk or cassette, and use it in your own programs whenever you wish.

Program 2 demonstrates how to use the subroutine, though I hope the comments make it pretty much self-explanatory. (Perhaps I should note that a command of R reads a sector, P writes a sector without verifying it, and W both writes and verifies a sector.) To use Program 2, you must add the subroutine from Program 1. You can either type in the lines from Program 1, or ENTER them from disk or tape if you have LISTed out a copy of Program 1. Program 3 is the source code behind the DATA statements in line 9210 of Program 1.

If you type in and use Program 2, you might like to remember that the *volume table of contents* (VTOC) of a DOS 2.0-compatible disk is in sector 360. The directory occupies sectors 361 to 368. Sectors 1, 2, and 3 are for booting only. All other sectors from 4 to 719 should be DOS file sectors. (See *COMPUTE! Books' Inside Atari DOS* for more info. Caution: The diagram of the sector link bytes is wrong.)

Finally, I give you a hint and challenge for next month: Most drives not made by Atari allow the user to specify their configuration (for example, single or double density). You can read their configuration blocks with an SIO command of N (or write via O). But be careful! DSIZE must be given as 12 bytes. Can you modify our subroutine to read the configuration block? Good luck.

DCB Layout Table

Location	Name	Size	Purpose
Hex	Dec		
300	768	DDEVIC	1
301	769	DUNIT	1
302	770	DCOMND	1
303	771	DSTATS	1
304	772	DBUF	2
306	774	DTIME	2
308	776	DBYTE	2
30A	778	DAUX	2

Program 1: SIO Subroutine

For instructions on entering this listing, please refer to "COMPUTE!'s Guide to Typing in Programs" published bimonthly in COMPUTE!.

```

LF 9000 REM .....
JG 9010 REM DISK SECTOR I/O
ROUTINE
JF 9020 REM . ENTER:
JQ 9030 REM .(3 SPACES)secto
r number in SECTOR
ND 9040 REM .(3 SPACES)drive
number in DRIVE
L 9050 REM .(3 SPACES)buffe
r address in ADDR
IP 9060 REM .(3 SPACES)comma
nd in CMD$
NJ 9070 REM .(3 SPACES)densi
ty in DENSITY
GM 9080 REM (only "R","W","P
" are valid for CMD$
)
EA 9090 REM (only 1=SGL and
2=DBL are valid for
DENSITY)
FA 9100 REM . EXIT:
CH 9110 REM .(3 SPACES)statu
s in SIOSTATUS
LA 9120 REM
OI 9160 TRAP 9220:REM activa
ted if SIOCALL$ alre
ady DIM'd
IO 9170 DIM SIOCALL$(16)
MC 9180 RESTORE 9210
JP 9190 FOR CNT=1 TO 14:READ
BYTE
EN 9200 SIOCALL$(CNT)=CHR$(B
YTE):NEXT CNT
MC 9210 DATA 104,32,89,228,1
73,3,3,133,212,169,0
,133,213,96
FB 9220 TRAP 40000:REM turn
off TRAP
MO 9230 POKE 768,ASC("1"):RE
M don't ask me why
GC 9240 POKE 769,DRIVE:REM m
ust be 1 through 8
OJ 9250 POKE 770,ASC(CMD$)
DN 9260 POKE 771,128:REM ass
ume write
270 IF CMD$="R" THEN POK
E 771,64
HA 9280 POKE 773,INT(ADDR/25
6):REM buffer address

```

```

PF 9290 POKE 772,ADDR-256*PE
EK(773)
FB 9300 POKE 774,3:REM short
timeout
JK 9310 POKE 775,0:REM (high
byte of timeout)
AA 9320 POKE 776,128:POKE 77
7,0:REM assume singl
e density
LG 9330 IF DENSITY=2 THEN PO
KE 776,0:POKE 777,1
KK 9340 POKE 779,INT(SECTOR/
256)
LD 9350 POKE 778,SECTOR-256*
PEEK(779)
HM 9360 SIOSTATUS=USR(ADR(SI
OCALL$))
LD 9370 RETURN

```

Program 2: SIO Demo

For instructions on entering this listing, please refer to "COMPUTE!'s Guide to Typing in Programs" published bimonthly in COMPUTE!.

```

KC 1000 REM PROGRAM TO DEMON
STRATE SECTOR READ S
UBROUTINE
HJ 1010 REM NOTE: rather tha
n ask questions, we
EB 1020 REM .(5 SPACES)assum
e that we will work
with drive
KP 1030 REM .(5 SPACES)numbe
r 1 and that it is s
ingle
HK 1040 REM .(5 SPACES)densi
ty (128 byte sectors
)
KK 1050 REM
PA 1100 DIM BUFFER$(256):REM
guaranteed adequate
ML 1110 ADDR=ADR(BUFFER$):RE
M required by subrou
tine
PI 1120 DRIVE=1:REM assumpti
on...easily changed
HC 1130 DENSITY=1:REM assump
tion...ditto
JO 1140 DIM CMD$(1):CMD$="R"
:REM always, for thi
s demo
KL 1150 REM
NB 1160 PRINT "What sector t
o display";
CJ 1170 INPUT SECTOR
DD 1180 GOSUB 9000

```

```

EN 1190 GRAPHICS 0
DL 1200 PRINT "Read Sector "
;SECTOR;" gave Statu
s ";SIOSTATUS
OP 1210 SIZE=DENSITY*128:REM
size is 128 or 256
CJ 1220 SECTOR=PEEK(ADDR+SI
Z E-3)
JC 1230 FILE=INT(SECTOR/4)
EP 1240 SECTOR=SECTOR-4*FILE
ON 1250 SECTOR=SECTOR*256+PE
EK(ADDR+SIZE-2)
EA 1260 CNT=PEEK(ADDR+SIZE-1
)
DO 1270 PRINT "If DOS file s
ector, this is file
#";FILE
NB 1280 PRINT " there are "
;CNT;" bytes in this
sector"
NA 1290 PRINT " and the nex
t sector is number "
;SECTOR
FB 1300 PRINT
JL 1310 FOR LINE=0 TO DENSIT
Y*128-1 STEP 8
FP 1320 BYTE=LINE:GOSUB 1500
:PRINT " ";
MK 1330 FOR CNT=0 TO 7
PD 1340 BYTE=PEEK(ADDR+LINE+
CNT):GOSUB 1500:PRIN
T " ";
ON 1350 NEXT CNT
NM 1360 FOR CNT=0 TO 7
DA 1370 BYTE=PEEK(ADDR+LINE+
CNT)
AD 1380 IF BYTE>127 THEN BYT
E=BYTE-128
BD 1390 PRINT CHR$(27);CHR$(
BYTE);
OJ 1400 NEXT CNT
FD 1410 PRINT
CO 1420 NEXT LINE
FF 1430 PRINT
MK 1440 GOTO 1160
LA 1450 REM .....
...
PF 1460 REM A QUICKY DECIMAL
TO HEX CONVERTER
MF 1500 TRAP 1520
DO 1510 DIM HX$(16):HX$=""012
3456789ABCDEF"
PD 1520 TRAP 40000
EK 1530 HX=INT(BYTE/16)+1:PR
INT HX$(HX,HX);:HX=B
YTE-16*HX+17:PRINT H
X$(HX,HX);
KK 1540 RETURN

```

Program 3: Subroutine Source Code

Note: This listing is provided for informational purposes; it requires an assembler to enter into your computer.

```

*= anyplace
CALLSIO
PLA ;throw away count
; of arguments
JSR SIOV ;at $E459)
LDA DSTATS ;SIO status
; (from DCB)
STA FR0 ;floating point
; register 0, $D4
LDA #0
STA FR0+1 ;(to get a two-
; byte value)
RTS ;back to BASIC caller

```



Atari Disk Drive Compatibility

Way back in 1978, when Atari announced the double-density 815 disk drive, Percom Data Corporation saw the prototypes displayed at several shows and decided it could easily build a better drive which would sell for less.

Because Percom produced both single- and double-sided disk drives using both single and double density, and because it wanted to maintain compatibility with both the single-density 810 and double-density 815 drives, Percom invented the *configuration block* (more on this below). With some cooperation from a small, brand-new software company (wonder who that could be) which had inherited the source code rights to Atari's File Management System (FMS), Percom succeeded in establishing standards which have been adhered to by all other Atari-compatible drive manufacturers. All Atari-compatible drive manufacturers except one, that is: Atari. Before the 815 even hit the market, Atari dropped it from the product line. Years later, in 1984, Atari introduced the "enhanced density" 1050, which is actually somewhere between single- and double-density. Sigh.

As of this writing, the following drives and/or modification kits are known to be capable of understanding the Percom-standard double-density mode and configuration table: Percom, Indus, Amdek, Astra, Trak, Rana, SWP (ATR-8000), Happy Doubler, and ICD's US Doubler.

The Percom Config Block

As defined by the Percom standard, a config block is a set of 12 bytes within the memory of the disk control microprocessor—which is inside your disk drive(s). You read a drive's config block by passing "N" to it as an SIO command. You can write a new config block to a drive via an "O" command. The "N" and

"O" commands closely parallel the "R" and "W" sector input/output commands, except the data length is always 12 bytes and no sector number is needed. The 12 bytes in the block are shown in the table.

Byte #	# of Bytes	Description
0	1	Number of Tracks
1	1	Step Rate
2-3	2	Sectors per Track
4	1	Number of Sides or Heads
5	1	Density (0=Single, 4=Double)
6-7	2	Bytes per Sector
8	1	Drive Selected?
9	1	Serial Rate Control
10-11	2	Miscellaneous (reserved)

This table requires some explanation. First, all the double-byte values are in high-byte/low-byte order, the opposite of normal 6502 practice (because that's how the microprocessor Percom used in their drives worked). Also, not all these values have meaning to all manufacturers. In fact, some don't allow you to change more than two or three of the values listed here.

The Step Rate controls the speed of a drive's head stepping motor, and the values used here have no universal meaning. A step rate of 2 may mean 6 milliseconds per track to one drive, 20 milliseconds per track to another, or be illegal to yet another.

Number of Sides is actually one less than the actual number. So most drives use a zero here, meaning one head.

Changing the value of Drive Selected may turn the drive off as far as the computer is concerned. Percom must have had its reasons for this, but I don't know what they were.

Changing The Config Block

For the Density byte of the config block, I don't know of any drives which use values other than 0 (FM mode, single density) or 4 (MFM

mode, double density). If you find a drive that actually *uses* some other value (not just ignores it), let me know.

The Serial Rate Control value and Miscellaneous bytes have no universal meanings. Some drives will remember these values if you change them; other drives ignore your values.

So that leaves Number of Tracks, Sectors per Track, and Bytes per Sector, all of which should be self-explanatory. Again, though, many drives ignore values outside certain legal ranges. Indus drives, for example, reject any changes to the number of tracks or sectors. In fact, Indus pays attention only to the Bytes per Sector and the Density bytes. Experiment with your own drive(s). See what they will and will not allow. And even if they seem to allow a change, do they execute it or ignore it? (Fun, if you're a masochist, right?)

And just how do you read and/or change the config block? Have a look at the BASIC program following this column. It should be pretty much self-explanatory. You can use the subroutines at 8010, 8210, and 9010 in your own programs. Remember what we said at the beginning, however: Atari drives do not follow the Percom config block standard. As a result, *this program works only on Atari-compatible disk drives, not on the Atari 810 or 1050.*

Configuration Block Modifier

For instructions on entering this listing, please refer to "COMPUTE!'s Guide to Typing In Programs" published bimonthly in COMPUTE!.

```
K0 1010 REM
N0 1020 REM CONFIGURE FROM B
      ASIC
K1 1030 REM
N1 1050 DIM TEMP$(20), TBL$(1
      2), CMD$(1)
G0 1060 GRAPHICS 0:PRINT "
      *** DISK CONFIGURATI
      ON PROGRAM ***"
N0 1070 PRINT :PRINT :PRINT
```

```

"what disk drive will we work with";
ND 1080 INPUT DRIVE
EA 1090 IF DRIVE<1 OR DRIVE>8 OR DRIVE<>INT(DRIVE) THEN RUN
NL 1100 GRAPHICS 0:PRINT "DRIVE #";DRIVE;
AL 1110 GOSUB 8000
PB 1120 IF SIOSTATUS<128 THEN 1170
DI 1130 PRINT "won't let me read"
IJ 1140 PRINT "{3 SPACES}the configuration block"
OI 1150 PRINT:PRINT "It gave me error #";SIOSTATUS
AO 1160 STOP
CA 1170 PRINT " looks like this:";PRINT
CI 1180 PRINT TRACKS;" TRACKS of ";SECTORS/PERTRACK;" SECTORS each"
JB 1190 PRINT:PRINT "each sector has ";BYTESPERSECTOR;" BYTES, & density"
NB 1200 PRINT " is ";DENSITY;" considered ";
NO 1210 IF DENSITY=0 THEN PRINT "SINGLE density,"
NM 1220 IF DENSITY=4 THEN PRINT "DOUBLE density,"
DN 1230 IF DENSITY<>0 AND DENSITY<>4 THEN PRINT "UNKNOWN DENSITY,"
JN 1240 PRINT " with ";SIDE$;" SIDE(S).";
IG 1250 PRINT:PRINT "the STEP RATE setting is ";STEPRATE
AI 1260 PRINT "other settings are SELECT=";SELECT;";"
NM 1270 PRINT " ACIA=";ACIA;"; and MISC=";MISC
EJ 1280 PRINT:PRINT "SELECT A CHOICE:"
DH 1290 PRINT "{3 SPACES}0 - quit and save configuration"
DF 1300 PRINT "{3 SPACES}1 - change drive setting(s)"
HP 1310 PRINT "{3 SPACES}2 - work with another drive"
EI 1320 PRINT:PRINT "your choice ";:INPUT CHOICE
GK 1330 IF CHOICE=0 THEN JUNK=USR(58484)
JF 1340 IF CHOICE=2 THEN RUN
ED 1350 GRAPHICS 0:PRINT "Enter new values. Hit RETURN to"
AB 1360 PRINT " leave a value unchanged."
FI 1370 PRINT
BF 1380 PRINT "TRACKS";:TEMP=TRACKS
N 1390 GOSUB 7000:TRACKS=TEMP
N 1400 PRINT "SECTORS PER TRACK";:TEMP=SECTORS/PERTRACK
N 1410 GOSUB 7000:SECTORS/PERTRACK=TEMP

```

```

PM 1420 PRINT "BYTES PER SECTOR";:TEMP=BYTESPERSECTOR
PK 1430 GOSUB 7000:BYTESPERSECTOR=TEMP
NA 1440 PRINT "NUMBER OF SIDES";:TEMP=SIDES
DG 1450 GOSUB 7000:SIDES=TEMP
NE 1460 PRINT "DENSITY";:TEMP=DENSITY
DA 1470 GOSUB 7000:DENSITY=TEMP
FK 1480 PRINT
FH 1490 PRINT "STEP RATE";:TEMP=STEPRATE
CC 1500 GOSUB 7000:STEPRATE=TEMP
AA 1510 PRINT "SELECT";:TEMP=SELECT
HM 1520 GOSUB 7000:SELECT=TEMP
JO 1530 PRINT "ACIA";:TEMP=ACIA
MM 1540 GOSUB 7000:ACIA=TEMP
MA 1550 PRINT "MISCELLANEOUS WORD";:TEMP=MISC
DM 1560 GOSUB 7000:MISC=TEMP
BH 1570 GOSUB 8200
PE 1580 IF SIOSTATUS<128 THEN 1100
CD 1590 PRINT:PRINT
GI 1600 PRINT "Unable to set that configuration!"
JH 1610 PRINT " drive issued error #";SIOSTATUS
BP 1620 PRINT:PRINT "(hit RETURN to continue)"
MF 1630 GOTO 1100
CH 7000 REM ENTER DATA OR NO CHANGE
LN 7030 PRINT "[";TEMP;"] ?";
LF 7040 INPUT TEMP$
BF 7050 IF LEN(TEMP$) THEN TEMP=VAL(TEMP$)
KN 7060 RETURN
IP 8000 REM EXTRACT INFO FROM TABLE
IN 8030 TBL=ADR(TBL$):ADDR=TEMP
MJ 8040 CMD$="N":GOSUB 9000:REM ---READ BLOCK---
IF 8050 TRACKS=PEEK(TBL+0)
CH 8060 STEPRATE=PEEK(TBL+1)
OI 8070 SECTORS/PERTRACK=PEEK(TBL+2)*256+PEEK(TBL+3)
JI 8080 SIDES=PEEK(TBL+4)+1
OG 8090 DENSITY=PEEK(TBL+5)
KJ 8100 BYTESPERSECTOR=PEEK(TBL+6)*256+PEEK(TBL+7)
IC 8110 SELECT=PEEK(TBL+8)
NC 8120 ACIA=PEEK(TBL+9)
PA 8130 MISC=PEEK(TBL+10)*256+PEEK(TBL+11)
KN 8140 RETURN
FP 8200 REM PUT NEW INFO INTO TABLE
IO 8230 TBL=ADR(TBL$):ADDR=TEMP
CO 8240 POKE TBL+0,TRACKS
MA 8250 POKE TBL+1,STEPRATE
PB 8260 POKE TBL+2,INT(SECTORS/PERTRACK/256)
JE 8270 POKE TBL+3,SECTORS/PERTRACK-PEEK(TBL+2)*256
EE 8280 POKE TBL+4,SIDES-1
JA 8290 POKE TBL+5,DENSITY

```

```

KP 8300 POKE TBL+6,INT(BYTESPERSECTOR/256)
FB 8310 POKE TBL+7,BYTESPERSECTOR-PEEK(TBL+6)*256
CN 8320 POKE TBL+8,SELECT
NM 8330 POKE TBL+9,ACIA
MM 8340 POKE TBL+10,INT(MISC/256)
JO 8350 POKE TBL+11,MISC-PEEK(TBL+10)*256
DO 8360 CMD$="O":GOSUB 9000:REM ---WRITE BLOCK---
LC 8370 RETURN
ME 9000 REM DISK DENSITY CHANGE ROUTINE
LA 9030 REM
LN 9040 REM ENTER: DRIVE NUMBER IN DRIVE
DC 9050 REM "{5 SPACES}buffer address in ADDR"
IP 9060 REM "{12 SPACES}command in CMD$"
LE 9070 REM
GH 9080 REM (ONLY "N" AND "O" ARE VALID FOR CMD$)
LG 9090 REM
GM 9100 REM EXIT: status in SIOSTATUS
KP 9110 REM
OL 9130 TRAP 9190:REM activated if SIOCALL$ already DIM'd
IL 9140 DIM SIOCALL$(16)
NF 9150 RESTORE 9180
JN 9160 FOR CNT=1 TO 14:READ BYTE
FD 9170 SIOCALL$(CNT)=CHR$(BYTE):NEXT CNT
NI 9180 DATA 104,32,89,228,173,3,3,133,212,169,0,133,213,96
FH 9190 TRAP 40000:REM turn off TRAP
ML 9200 POKE 768,ASC("1");REM don't ask me why
FP 9210 POKE 769,DRIVE:REM must be 1 through 8
OG 9220 POKE 770,ASC(CMD$)
MA 9230 POKE 771,128:REM assume output
LI 9240 IF CMD$="N" THEN POKE 771,64
GN 9250 POKE 773,INT(ADDR/256):REM buffer address
PC 9260 POKE 772,ADDR-256*PEEK(773)
FH 9270 POKE 774,3:REM short timeout
KA 9280 POKE 775,0:REM (high byte of timeout)
BI 9290 POKE 776,12:POKE 777,0:REM assume std config block
HG 9300 SIOSTATUS=USR(ADR(SIOCALL$))
KN 9310 RETURN

```




INSIGHT: Atari

Bill Wilkinson

Deactivating BASIC

My coworkers and I have received many requests from owners of the Atari 600XL, 800XL, and 130XE for a simple way to turn off the BASIC built into those computers. Of course, the method recommended by Atari is to hold down the OPTION button when you boot the system. If you forget to do this when booting a program that doesn't require BASIC, the ROM-based BASIC occupies address space that costs you more than 8,000 bytes of RAM. There are other reasons for turning off BASIC as well. For instance, you might like to turn it off temporarily to gain extra memory while duplicating a few files or disks. These jobs take less time and fewer disk swaps if the computer can use the 8K of memory vacated by disabling BASIC. And avoiding a reboot or two can save time, too.

Our solution is a pair of short machine language programs that let you turn BASIC on and off from DOS. (Note that they can't turn off a BASIC cartridge—or any other cartridge, for that matter—so they serve no purpose on the Atari 400, 800, and 1200XL computers.) Atari manuals suggest that turning off the built-in BASIC is as simple as changing one bit in the XL/XE memory control location (which used to control joystick ports 3 and 4 in the 400 and 800). That may be true if you're writing a machine language program that takes over complete control of the computer, but in many cases it doesn't work.

First, whenever you press the RESET button, the operating system restores the built-in BASIC to the state in which you booted it. Second, if you're using ordinary graphics mode screens (without a custom display list, etc.), the screen handler doesn't use the memory freed by removing BASIC. It thinks you're still using a 40K machine.

Going the other way—turning on BASIC after booting without it—can be even messier. If you suddenly enable BASIC without doing something about the screen, you'll find yourself staring at garbage as BASIC blithely wipes out the display list, screen memory, and perhaps more. Fortunately, all of these problems can be solved by following these few steps:

1. Turn the built-in BASIC off or on.
2. Tell the operating system you did so.
3. Change the master top-of-RAM pointer.
4. Close channel 0, the screen editor.
5. Reopen channel 0.

We can tell the operating system we changed the state of BASIC via the flag in memory location 1016 (\$3F8). The master top-of-RAM pointer is RAMTOP at location 106 (\$6A). Channel 0 is closed and reopened to force the screen driver to use the highest available memory. Don't worry if that sounds a bit arcane. The program listed here automatically creates two machine language programs that do all the work for you. Be sure to save a copy before you run it.

```

GD 100 DIM NAME$(20)
NM 110 LINE=800:GOSUB 210
NO 120 LINE=900:GOSUB 210
BL 130 END
DE 210 CHECK=0:RESTORE LINE
GM 220 FOR CNT=1 TO 57:READ
    BYTE
EL 230 CHECK=CHECK+BYTE:NEXT
    CNT
EI 240 READ TEST:IF CHECK<>T
    EST THEN STOP
WJ 250 READ NAME$:OPEN #1,8,
    0,NAME$
DE 260 RESTORE LINE
GM 270 FOR CNT=1 TO 57:READ
    BYTE
KF 280 PUT #1,BYTE:NEXT CNT
BF 290 CLOSE #1
ND 300 RETURN
IN 810 DATA 255,255,0,4,44,4
    ,173,1,211,9,2,141,1
    
```

```

DL 830 DATA 211,169,1,141,24
    8,3,169,12,32,24,4
OG 840 DATA 169,192,133,106,
    169,3,141,66,3,169,42
ID 860 DATA 141,68,3,169,4,1
    41,69,3,162,0,76,86
OK 870 DATA 228,69,58,0,226,
    2,227,2,0,4
HM 880 DATA 5045,D:BASICOFF.
    COM
MF 910 DATA 255,255,0,4,44,4
    ,173,1,211,41,253,141
JI 930 DATA 1,211,169,0,141,
    248,3,169,12,32,24,4
OC 940 DATA 169,160,133,106,
    169,3,141,66,3,169,42
ID 950 DATA 141,68,3,169,4,1
    41,69,3,162,0,76,86
OL 970 DATA 228,69,58,0,226,
    2,227,2,0,4
EG 980 DATA 5295,D:BASICON.C
    OM
    
```

The program writes two binary files to disk on drive 1, naming them BASICON.COM and BASICOFF.COM. The first turns BASIC on and the second turns it off. To use either of them from DOS, simply choose the L (load binary file) option and enter the filename when prompted. (OS/A+ and DOS XL users need only type BASICON or BASICOFF in response to the D1: prompt.)

The next time you need to duplicate a disk or large file, load BASICOFF.COM first, copy the disk or file, then load BASICON.COM to reactivate BASIC. You'll save time, especially on a single-drive system. If you're writing machine language programs, call BASICOFF as a subroutine when you start your program. ©



The Hidden Power Of Atari BASIC

This month we're going to look at good old Atari BASIC. For once, though, I'm not going to talk about its problems. Instead, I'm going to tell you about a few of its many virtues. If you've been reading my column since it first appeared in the September 1981 issue of *COMPUTE!*, then some of this may seem repetitive; but it's time to introduce newcomers to some of this material.

Unfortunately, I am beginning to see more and more poorly written Atari BASIC programs. Generally, what happens is that someone not too well-versed in Atari BASIC attempts to translate a program from another computer's BASIC and botches the job. The last straw, for me, was a recently released book which is full of CAI (Computer Assisted Instruction) programs. All the programs do much the same thing, and all the programs are...well, just a lot of work for so little value.

Now, I'm all for using a computer for drill and practice, even though most of the educational programs which do this are dull and unimaginative (and often overpriced). But even the plainest of CAI programs can at least free up a teacher or parent for 20 or 30 minutes while a student is checking his or her knowledge. And if all you want your CAI program to do is ask questions and wait for a response, then all such programs can be essentially the same. So that's what I'm going to give you this month: a "formula" program for drill and practice.

I also mentioned that we would look at some of the virtues of Atari BASIC, so let's do that first. Among microcomputer BASICs, Atari BASIC is nearly unique in its flexibility in the use of GOTO, GOSUB, and RESTORE. Specifically, each of these statements accept any numeric expression as the line

number they reference. Combined with Atari BASIC's variable-name flexibility, this means you can code such oddities as:

```
GOSUB CALCULATEGROSSPAY
```

and

```
RESTORE 20000+10*CURRENTROOM
```

Most Atari BASIC books refer to these capabilities briefly, if at all. But there is some real hidden power here, as we are about to find out. Rather than belabor the point, let's take a look at the accompanying listing and analyze it a step at a time.

Using Variables As Labels

Line 1010 is fairly obvious, so let's start with lines 1060 to 1080. The variables being set here are actually going to be used as labels, the targets of GOTO and GOSUB statements. The only thing you have to be careful of with this method is renumbering—some renumbering utilities warn you when they encounter a variable being used as a label, and some don't.

Now, after setting the DATA pointer in line 1090, we get a line of DATA, assigning the first byte to the variable TYPE\$. The action we take next depends on what type of line we got. We use an exclamation point to indicate a screen clear is needed, a colon for an extra blank line, and a period to flag an ordinary text line. In any of these cases, we print the rest of the line and get another one. If the type is an asterisk, the program halts. If the type is a question mark, then it's time for the student to answer.

At this time, let's look at the DATA in lines 10000-10003. The first line begins with an exclamation point, so the screen is cleared and it is printed. Then the colon asks for a blank line before the next line is displayed. Finally, the question mark tells the program to ask

for a response. But what's the rest of that funny stuff: 1=Y,0,10010?

Back at lines 1200-1260, you can see that the digit (a 1 in line 10002) tells the number of possible answers to the question, and the next character indicates the type of answer which is acceptable (the equal sign here asks for an exact match). The program then prompts the user for an answer (the #16 suppresses the INPUT prompt) and prepares to test its validity. The loop in 1310-1360 checks each valid answer against the user's response.

If an exact answer is needed, even the length of the answer counts. (Example: In line 10002, we have allowed only a single exact answer, the letter Y.) Another flag indicates whether the valid answer can be found somewhere in the user's response line. Line 10012, for example, passes any answer containing the word GRANT (such as MIGRANT WORKERS), so some care is needed in using this type. Finally, if none of the valid answers matches the user's response, the program falls through to lines 1400-1420.

So far, all this has been very straightforward, and it would work on almost any BASIC. Now comes the tricky stuff. Look at line 1320, where we READ numbers into the variables GOSUBLINE and DATA-LINE. What we're doing is establishing an action to take and a new set of DATA to access if the user's response matches a valid answer. Similarly, in line 1420 we read values to be used if no valid answer is given. Finally, the "magic" of this program is revealed in lines 1510 and 1520.

If we READ a number other than zero for GOSUBLINE, the program actually GOSUBs to that number. And, in any case, we change the DATA pointer to the

new DATALINE. If you can't predict what happens if you answer DUCK to the second question (because of the DATA in lines 10012-10014), please type this program and try it out.

Now, the real beauty of this program is that it works with almost any kind of question and answer session. It allows for multiple choice questions (use a format like ?3=,A,0,100,B,0,100,C,0,200), true/false, and so on. It provides for special help if needed (via the GOSUBLINES). And, last but by no means least, it is expandable. You could add many different statement types, question types, or whatever quite easily. And it's all made possible thanks to Atari BASIC.

Multiple Choice Quiz

```
DL 1000 REM === INITIALIZATI
ON ===
MF 1010 DIM LINE$(120), ANS$
(20), TYPE$(1)
ID 1060 INEXACT=2000:EXACT=2
100
NI 1070 MAINLOOP=1100:QUESTI
ON=1200
MC 1080 MATCHED=1500
CC 1090 RESTORE 10000:REM wh
ere we start
PL 1100 REM === THE MASTER L
OOP ===
FE 1110 READ LINE$:TYPE$=LIN
E$
FJ 1120 IF TYPE$="?" THEN GO
TO QUESTION
DM 1130 IF TYPE$="!" THEN PR
INT CHR$(125);
DC 1140 IF TYPE$=":" THEN PR
INT
GN 1150 IF TYPE$="*" THEN EN
D
CE 1160 PRINT LINE$(2)
GO 1170 GOTO MAINLOOP
ON 1200 REM === PROCESS A QU
ESTION ===
JI 1210 QCNT=VAL(LINE$(2,2))
DI 1220 TYPE$=LINE$(3)
PL 1230 POSITION 2,20
KG 1240 PRINT CHR$(156);CHR$
(156);
DC 1250 PRINT "Your Answer ?
";
FL 1260 INPUT #16,LINE$
OG 1300 REM === PROCESS THE
ANSWER ===
MF 1310 FOR ANS=1 TO QCNT
CX 1320 READ ANS$,GOSUBLINE,
DATALINE
BL 1330 IF TYPE$="*" THEN GO
SUB INEXACT
JP 1340 IF TYPE$=":" THEN GO
SUB EXACT
1350 IF MATCH THEN GOTO M
ATCHED
OL 1360 NEXT ANS
EI 1400 REM === ANSWER DOESN
'T MATCH ===
JA 1410 REM (read error cond
itions and fall thru
)
```

```
PJ 1420 READ GOSUBLINE,DATAL
INE
NO 1500 REM === ANSWER MATCH
ED ===
FF 1510 IF GOSUBLINE THEN GO
SUB GOSUBLINE
CO 1520 RESTORE DATALINE
GB 1530 GOTO MAINLOOP
LD 2000 REM === INEXACT MATC
H ROUTINE ===
BK 2010 MATCH=0:ALEN=LEN(ANS
$)
GB 2020 SIZE=LEN(LINE$)-ALEN
+1
AL 2030 IF SIZE<1 THEN RETUR
N
BH 2040 FOR CHAR=1 TO SIZE
GL 2050 IF LINE$(CHAR,CHAR+A
LEN-1)=ANS$ THEN MAT
CH=1:RETURN
CF 2060 NEXT CHAR
KJ 2070 RETURN
BN 2100 REM === EXACT MATCH
ROUTINE ===
EO 2110 MATCH=(ANS$=LINE$)
KF 2120 RETURN
KK 10000 DATA !Ready to try
out this program?
KP 10001 DATA : (answer Y o
r N)
LJ 10002 DATA ?1=,Y,0,10010
FL 10003 DATA 0,10000
PJ 10010 DATA !A tribute to
Groucho Marx:
CK 10011 DATA :Who is buried
in Grant's tomb?
MH 10012 DATA ?2#,GRANT,0,10
040
HF 10013 DATA DUCK,10020,100
30
CJ 10014 DATA 10050,10060
OH 10020 REM special sound r
outine
KI 10021 FOR FREQ=120 TO 20
STEP -10
CG 10022 FOR VOLUME=15 TO 0
STEP -0.5
JK 10023 SOUND 0,FREQ,10,VOL
UME
LF 10024 NEXT VOLUME:NEXT FR
EQ
NI 10025 RETURN
LB 10030 DATA !You said the
secret word!
LH 10031 DATA :You win $100.
DK 10032 DATA *
KL 10040 DATA !Great! You g
et the consolation
DATA .prize of $50.
DL 10042 DATA *
LE 10050 REM raspberry
FK 10051 FOR VOLUME=15 TO 0
STEP -0.25
PK 10052 SOUND 0,4,80,VOLUME
:NEXT VOLUME
NJ 10053 RETURN
KM 10060 DATA !Sorry. You l
ost.
DM 10061 DATA *
```

COMPUTE!

**TOLL FREE
Subscription
Order Line
800-334-0868
In NC 919-275-9809**

**Copies
of articles
from this
publication
are now
available
from the
UMI Article
Clearinghouse.**

For more information
about the Clearinghouse,
please fill out and mail back
the coupon below.

**UMI Article
Clearinghouse**

Yes! I would like to know more about UMI
Article Clearinghouse. I am interested in
electronic ordering through the following
system(s):

☐ DIALOG/Dialorder ☐ ITT Dialcom
☐ OnTyme ☐ OCLC ILL
Subsystem

☐ Other (please specify) _____
☐ I am interested in sending my order by
mail.

☐ Please send me your current catalog and
user instructions for the system(s) I
checked above.

Name _____

Title _____

Institution/Company _____

Department _____

Address _____

City _____ State _____ Zip _____

Phone (____) _____

Mail to: University Microfilms International
300 North Zeeb Road, Box 91 Ann Arbor, MI 48106



INSIGHT: Atari

Bill Wilkinson

Avoiding Memory Confusion In Atari BASIC

After a couple of months of standing on my soap box, I've decided to step off and get back to business again. Before I do, though, here's one more little rant and rave: I can now express my opinion of Atari's new BASIC for the 520ST. In a word: disappointing. Neither ST Logo nor ST BASIC are viable production languages, which means you can't write commercial applications with them. Since even the C compiler included in Atari's \$300 software developer's package doesn't support double-precision arithmetic, limiting you to six decimal digits of precision, you'd better be ready to purchase some language from an outside vendor if you're serious about doing any programming on the ST machines.

Several months ago, I asked all you loyal readers to send me a postcard or letter giving ratings to the best or worst Atari-oriented books. Although I was a little underwhelmed by the response, I did get enough ballots to at least select the three favorites. Among these three, however, there was no clear-cut winner. And I happen to feel that is appropriate, since each addresses a different part of the knowledge an Atari programmer needs. Anyway, according to my readers, the best books are (drum roll...the envelope please): *The ABC's of Atari Computers*, by Dave Mentley, published by Datamost; *Your Atari Computer*, by Lon Poole et al, published by Osborne/McGraw Hill; and *Mapping the Atari*, by Ian Chadwick, published by COMPUTE! Books. (Incidentally, you may have noticed that COMPUTE! Books has been shipping the new, revised version of *Mapping the Atari*, which has several appendices and notes devoted to the XL and XE machines.)

The rest of this column responds to a number of reader re-

quests. Although the topic has been covered in COMPUTE! before (at least in part), there are many newcomers out there. And even if you aren't a newcomer, maybe I can provide more insight into the concepts involved.

Finding Free Memory

Q: Where in memory can a programmer put machine language routines, character sets, player/missile graphics, and the like?

A: There is no simple answer, because it depends on which language you're using, which DOS, etc. A couple of years ago, I did an entire series on relocatable machine language which was related to this problem. So this time, let's tackle a simpler and more specific question: Where can I put a custom character set? The following techniques will also work for many other uses, including player/missile graphics.

When allocating memory, Atari BASIC—as well as BASIC XL and BASIC XE—looks at and believes the contents of two memory locations, LOMEM and HIMEM (located at \$2E7, decimal 743, and \$2E5, decimal 741, respectively). BASIC always starts your program where LOMEM tells it to and lets it grow as high as the value in HIMEM. Remember that this “growing” includes not just your BASIC code, but also the strings and arrays dimensioned by your program. Let's consider LOMEM first.

The fact that a program always starts at LOMEM implies that if we increase the value of LOMEM and then load a program, the memory between the old value and the new one is available for whatever purposes we have in mind. On the other hand, once a BASIC program is loaded into memory, it ignores changes to LOMEM. This means we can have one program change the contents of LOMEM and then

chain to another program. The first program is unaffected by the change, but the second will be loaded at the new LOMEM. Programs 1 and 2 demonstrate this technique.

Examine Program 1, which ensures that the memory we wish to reserve starts on a particular boundary. Remember that full character sets (128 characters) must start on 1K memory boundaries, and half sets must start on 512-byte boundaries. There are similar rules for player/missile graphics (see “Atari Animation With P/M Graphics,” a three-part series starting in the September 1985 issue of COMPUTE!). If you actually type in and run the programs below, you'll be in for a little surprise. But *do not* omit the REMark statements from Program 2, or you'll miss half the fun. Feel free to omit them from Program 1. For the programs to function properly, you must save Program 2 with the filename PROGRAM2.BAS (see line 900 of Program 1). If you're using cassette instead of disk, change line 900 in Program 1 to RUN “C:” and make sure the tape is cued to Program 2 before you run Program 1.

A minor caution: The reason we base the changes to LOMEM on the contents of locations 128 and 129 (BASIC's internal MEMLO pointer) instead of the actual LOMEM contents is complex. I have discussed it in this column before, but the heart of the problem is that some Atari device drivers (including the 850 Interface Module's R: handler) do not correctly restore LOMEM when the SYSTEM RESET button is pressed. After a reset, BASIC's pointer is more reliable. For the same reason, and for safety's sake, programs bumping LOMEM should always bump it higher than the top of the BASIC program currently in memory. And one last piece of advice: If you run Program

1 over and over again, it keeps raising LOMEM higher and higher. Eventually you'll run out of memory. You probably need some sort of flag elsewhere in memory (Page 6?) which tells the program not to raise LOMEM again.

Modifying HIMEM

Enough about LOMEM; what about HIMEM? Truthfully, if you know how big your program is and what it's going to use in the system, you can put anything you want (character sets, machine language, player/missile shape data, etc.) in the memory between the top of your program and the bottom of screen memory. The only time the contents of HIMEM are used is when BASIC checks to ensure that APPMHI (location 14, \$0E) hasn't collided with it. APPMHI is essentially BASIC's high water mark. It keeps track of the top of the runtime stack, which is always above the string and array space, which in turn is always above your program. So, if you *know* that your program, its data, and its stack will never grow too large, you could ignore HIMEM altogether. It's much cleaner, though, to tell the system what you're using by modifying HIMEM.

How and why does HIMEM change if you don't do this? The most usual cause is a change in the graphics mode. For example, while ordinary text screen graphics (GRAPHICS 0) occupy less than 1K of memory, several graphics modes (such as modes 8, 9, 10, 11, and 15) require 8K of screen memory. To demonstrate this, type in and run the following line, preferably after hitting the SYSTEM RESET button:

```
G0=FRE(0):GR.8:PRINT G0,FRE(0),G0-FRE(0)
```

This displays three numbers: memory available for your program(s) in text mode, usable memory in mode 8, and the extra amount used by mode 8 graphics.

Generally, the best method is to always put your own goodies below the area occupied by the most memory-intensive graphics mode you plan to use. So either look in a memory map book to find out how much room a certain graphics mode will take, or simply change modes before using the

memory.

For an example, try Program 3. It's essentially the same as Program 2. The difference is simply where we move the character set. The REMarks explain where you should insert your own graphics mode declaration.

For instructions on entering this listing, please refer to "COMPUTE!'s Guide to Typing in Programs" in this issue of COMPUTE!.

Program 1: MEMLO Bumper

```
HF 100 REM
DS 110 REM THIS PROGRAM IS U
SED TO
BH 120 REM RESERVE SIZE"PAGE
S" OF
TG 130 REM MEMORY FOR PROGRA
M2.BAS
HJ 140 REM
AE 150 REM (A "PAGE" IS 256
BYTES
HL 160 REM
CA 170 REM THIS PROGRAM ALSO
ENSURES
FL 180 REM THAT THE RESERVED
SPACE
LK 190 REM STARTS ON THE GIV
EN BOUNDARY
JD 200 REM (TO INSURE, FOR EX
AMPLE, THAT
OK 210 REM CHARACTER SETS ST
ART ON 1K
MB 220 REM BYTE BOUNDARIES)
HJ 230 REM
KB 500 SIZE=4:REM MUST BE AT
LEAST 4 PAGES (1024
BYTES)!
DI 510 BOUNDARY=4:REM ALSO G
IVEN IN PAGES
OC 520 IF PEEK(128)<>0 THEN
POKE 128,0:POKE 743,0
:SIZE=SIZE+1
MH 530 MEMLO=PEEK(129)+SIZE
BF 540 MEMLO=INT((MEMLO+BOUN
DARY-1)/BOUNDARY)*BOU
NDARY
AD 550 POKE 744, MEMLO
AM 560 POKE 129, MEMLO
JD 900 RUN "D:PROGRAM2.BAS"
```

Program 2: Character Set Mover, Version 1

```
DB 150 REM JUST AS A DEMO, T
HIS PROGRAM
KM 160 REM CHANGES THE CHAR
SET POINTER,
JJ 170 REM COPIES THE CHARAC
TER SET
GG 180 REM TO THE RESERVED M
EMORY,
BF 190 REM AND THEN RADOMLY
DESTROYS
FI 200 REM THE CHARACTERS!
HH 210 REM
HJ 220 REM HIT RESET TO QUIT
AND GET
LA 230 REM NORMAL CHARACTERS
AGAIN.
HK 240 REM
BI 250 GRAPHICS 0
JH 260 SIZE=4:REM SHOULD BE
THE SAME AS PROGRAM 1
```

```
BF 270 POKE 756, PEEK(129)-SI
ZE:REM CHBAS IS CHANG
ED
HA 280 BUFFER=PEEK(756)*256
FI 290 POKE 752,1:PRINT :REM
NO MORE CURSOR
NK 300 FOR ADDR=BUFFER TO BU
FFER+1023
NA 310 POKE ADDR,0:REM FIRST
CHANGE ALL CHARS
DO 320 NEXT ADDR:REM TO SAME
REPEATED PATTERN
CA 330 LIST 150,240:REM JUST
SOMETHING TO SHOW
IL 340 REM READY TO MOVE THE
CHARACTERS
HA 350 FOR ADDR=0 TO 1023
LH 360 POKE BUFFER+ADDR, PEEK
(57344+ADDR)
PE 370 NEXT ADDR
CB 380 REM MOVED...SLOWLY DE
STROYED
NG 390 POKE INT(RND(0)*1024)
+BUFFER, INT(RND(0)*25
6)
```

Program 3: Character Set Mover, Version 2

```
DB 150 REM JUST AS A DEMO, T
HIS PROGRAM
IA 160 REM CHANGES THE CHAR
SET POINTER
JJ 170 REM COPIES THE CHARAC
TER SET
GG 180 REM TO THE RESERVED M
EMORY,
BD 190 REM AND THEN RANDOMLY
DESTROYS
FI 200 REM THE CHARACTERS!
HH 210 REM
HJ 220 REM HIT RESET TO QUIT
AND GET
LA 230 REM NORMAL CHARACTERS
AGAIN.
HK 240 REM
BG 250 GRAPHICS 7:REM JUST T
O CLEAR ABOUT 4K OF M
EMORY!
HP 260 GRAPHICS 0:REM OR OTH
ER MODE
EF 270 SIZE=4
DH 280 REM ALWAYS DO FOLLOWI
NG AFTER THE GRAPHICS
STATEMENT
HJ 290 POKE 741,255:REM ENSU
RE END-OF-PAGE BOUN
DARY
MG 300 MEMHI=INT(PEEK(742)/S
IZE)*SIZE-SIZE
HB 310 POKE 742, MEMHI-1:REM
LOWER HIMEM
AH 320 POKE 756, MEMHI:REM CH
BAS IS CHANGED
GN 330 BUFFER=PEEK(756)*256
FE 340 POKE 752,1:PRINT :REM
NO MORE CURSOR
CC 350 LIST 150,240:REM JUST
SOMETHING TO SHOW
IN 360 REM READY TO MOVE THE
CHARACTERS
MC 370 FOR ADDR=0 TO 1023
LJ 380 POKE BUFFER+ADDR, PEEK
(57344+ADDR)
PG 390 NEXT ADDR
BK 400 REM MOVED...SLOWLY DE
STROYED
MP 410 POKE INT(RND(0)*1024)
+BUFFER, INT(RND(0)*25
6)
GE 420 GOTO 410
```

ton is pressed has an opcode of 124, so we POKE CONTRL,124. We must tell the VDI routine that no other parameters are being passed, so two more POKES are necessary: POKE CONTRL+2,0 and POKE CONTRL+6,0. Now we can call the VDI routine to read the mouse.

To read the horizontal and vertical position of the mouse, PEEK PTSOUT and PTSOUT+2, respectively. If the mouse button is pressed, PEEKing INTOUT will give a value of 1; otherwise, a zero is returned. (PTSOUT and INTOUT, like CONTRL, are also reserved variables for accessing VDI routines.)

The main loop of the piano program (line 30) simply waits until a mouse button is pressed. Once the button has been pressed, the vertical coordinates are checked to see if they are in the range of the piano keyboard (line 50). Then the vertical position is used to determine whether the key pressed is black or white (lines 50 and 60). If a black key is pressed, the note is calculated using the array B%; otherwise, the array W% is used.

Line 70 breaks the note value

down into note and octave and then, using the SOUND command, plays the note.

Line 80 sets the envelope shape to zero. This creates a note with a similar shape to a piano's envelope. Program execution is then sent back to the main loop to check the mouse button again and SOUND another note when it is pressed.

Program 1: Helicopter

```
10 for a=1000 to 643 step -2
20 wave 8,3,14,a
30 for td=1 to 100:next:next
40 for a=643 to 1000 step 2
50 wave 8,3,14,a
60 for td=1 to 100:next:next
70 sound 1,0:sound 2,0
```

Program 2: Ding

```
10 for a=1 to 12
15 sound 1,15,a,7
20 wave 1,1,14,5,1
30 for td=1 to 100:next:next
40 goto 10
```

Program 3: Piano

```
10 dim b%(16),w%(16)
20 gosub DRAWSCREEN:gosub
SETARRAY
```

```
30 gosub READMOUSE:if button=0
then 30
40 if y<70 or y>120 then 30
50 if y<100 then n=b%((x-16)/16.25)
60 if y>99 then n=w%((x-4)/16.25)
70 sound 1,15+15*(n=0),n-12*int((n-1)/12),3+int((n-1)/12)
80 wave 1,1,0,10000:goto 30
90 READMOUSE: poke contrl,124
100 poke contrl+2,0:poke contrl+6,0
110 vdisys(0)
120 x=peek(ptsout):y=peek(ptsout+2)
130 button=peek(intout)
140 return
150 DRAWSCREEN: color 1,1,1,1:fullw
2:clearw 2
160 for a=50 to 100 step 50
170 linef 20,a,280,a:next
180 for a=20 to 280 step 16.25
190 linef a,50,a,100:next
200 for a=1 to 11:read s
210 gosub 250:next:return
220 data 32.5,48.75,81.25,97.5
230 data 113.75,146.25,162.5
240 data 195,211.25,227.5,260
250 linef s,50,s,78
260 linef s,78,s+8,78
270 linef s+8,78,s+8,50
280 fill s+1,51:fill s+5,51
290 return
300 SETARRAY: for a=1 to 16:read
w%(a):next
310 for a=1 to 16:read b%(a):next
320 return
330 data 1,3,5,6,8,10,12,13
340 data 15,17,18,20,22,24
350 data 25,27
360 data 2,4,0,7,9,11,0,14,16
370 data 0,19,21,23,0,26,0
```



INSIGHT: Atari

Bill Wilkinson

Atari Character Codes

Last month's discussion about where and how to place things in memory served as a good lead-in to this month's topic: character codes. If you've read the hefty reference material, including COMPUTE! Book's *Mapping the Atari*, you may have discovered that your eight-bit Atari computer actually uses three different types of codes to represent the various characters (letters, numbers, punctuation, graphics symbols) it works with. All of these codes assign a unique number to represent each character, but the three codes are incompatible with each other because they use different numbering schemes.

The most commonly encountered code is called *ATASCII*, which

stands for ATari-version American Standard Code for Information Interchange. Except for the so-called *control characters*—such as carriage return, tab, and so on—ATASCII is compatible with standard ASCII. (Why Atari chose to modify the standard is anyone's guess.) ATASCII is the character code used by PRINT, INPUT, CHR\$(), ASC(), and most external devices such as printers and modems.

For example, in ATASCII (and ASCII), the code for uppercase A is 65. You can verify this in BASIC:

```
PRINT CHR$(65)
or
PRINT ASC("A")
```

Virtually every Atari BASIC book (even Atari's own) shows the

character represented by each ATASCII code. You can also run Program 1 below to display each character and its code. (Press CTRL-1 to pause and continue the display.)

Screen Codes

The second character code found in your Atari is the *keyboard code*. The keyboard code for any character is actually the value read from a hardware register in memory when the key for that character on the keyboard is pressed. Program 2 below lets you find the keyboard code for any character. Just for fun, try some of the keys or key combinations which don't normally produce characters, such as CTRL-SHIFT-

CAPS). Neat, huh?

Finally: *screen codes*. This term refers to the byte value you must store in memory to display the desired character on the screen. "What?" you ask, "How do those differ from the ATASCII codes?" After all, to put the string BANANA PICKLE PUDDING on the screen, all it takes is a simple BASIC statement:

```
PRINT "BANANA PICKLE PUDDING"
```

And besides, aren't the characters in quotes supposed to be ATASCII codes? Good questions. Now for some complicated answers.

Actually, if the original Atari designers had thought just a little harder and added just a few more logic gates to the thousands already in the ANTIC and GTIA chips, ATASCII and screen codes could have been one and the same. It's similar to the mistake of making ATASCII incompatible with ASCII. Sigh. But we're stuck with what we've got, so let's figure out how it works.

For starters, consider GRAPHICS 1 and GRAPHICS 2, the large-size character modes. You may have noticed that in either of these modes you can display only 64 different characters on the screen. Now, if you recall last month's demo programs, note that we can specify the *base address* of the character set. That is, we can tell ANTIC where the character set starts by changing the contents of memory location 756 (which is actually a *shadow register* of the hardware location which does the work—see *Mapping the Atari* for more on this).

In a sense, the ANTIC chip is fairly simplistic. When it finds a byte in memory which is supposed to represent a character on the screen, it simply adds the value of that byte (multiplied times eight, because there are eight bytes in the displayable form of a character) to the character set base address. This points to the memory address for that particular character. Except...well, let's get to that in a moment.

Exception To The Rule

Because we want GRAPHICS 1 and 2 (with their limited sets of 64 different characters) to display num-

bers and uppercase letters (omitting lowercase letters and graphics), for these two modes it makes sense that the character set starts with the dot representation of the space character and ends with the underline—codes 32 through 95, respectively.

But why are these 64 characters the only ones available in GRAPHICS 1 or 2? Because the upper two bits of a screen byte in these modes are interpreted as *color* information, *not* as part of the character (see the modification to Program 3 below). So only the lower six bits choose a character from the character set. Six bits can represent only 64 possible combinations, which is why these modes can display only 64 characters. Bit pattern 000000 becomes a space, 100101 is an E, and 111111 becomes an underline, and so on.

When you use GRAPHICS 0 (normal text), however, there is a strange side effect. In this mode, only the single upper bit is the color bit (actually, it's the inverse video bit). This leaves 7 bits to represent a character, so we can have values from 0 to 127 decimal (0000000 to 1111111 binary, \$00 to \$7F hex). Again, this value—after being multiplied by eight—is added to the value of the character set base address. But which numbers in that 0 to 127 range represent which characters?

Well, we already know what the first 64 characters are—since the Atari's hardware limitations dictate that they *must* be the same as in modes 1 and 2. So the next 64 are the other characters. Program 3 illustrates how the ATASCII character set is linked to the screen set. Note how all the characters are presented twice, once in screen code (i.e., character ROM) order and once in ATASCII order. For some additional fun and info on modes 1 and 2, change line 10 to GRAPHICS 1. (Do not change it to GRAPHICS 2 unless you put a STOP in line 65 after the first FOR-NEXT loop.) Do you see what I mean about the upper two bits being color information?

Now you know why there are three different character codes used in your computer. How can you take advantage of this information?

Well, if you combine this knowledge with the programs I presented last month, you could invent your own character set and design a word processor for some foreign language. (If you come up with a good Cyrillic character set, let me know.)

Actually, if you own an XL or XE machine, you have a second character set already built in. Just add this line to Program 3:

```
20 POKE 756,204
```

This tells the operating system and ANTIC that the base of the character set is at \$CC00, which is where the international character set resides. Someday you might find some use for these characters. How will you know until you try?

For instructions on entering these listings, please refer to "COMPUTE!'s Guide to Typing in Programs" in this issue of COMPUTE!.

Program 1: ATASCII Codes

```
OC 10 GRAPHICS 0
HK 20 FOR I=0 TO 255:PRINT I
ID 30 IF I=155 THEN PRINT "[
RETURN]":GOTO 50
IC 40 PRINT CHR$(27);CHR$(I)
ON 50 NEXT I
NH 60 REM USE CONTROL-1 TO P
AUSE
```

Program 2: Keyboard Codes

```
BC 10 DIM HEX$(16):HEX$="012
3456789ABCDEF"
PK 20 POKE 764,255
LL 30 KEYCODE=PEEK(764)
BC 40 IF KEYCODE=255 THEN 30
HF 50 HI=INT(KEYCODE/16):LOW
=KEYCODE-16*HI
LJ 60 PRINT "KEYCODE: HEX ";
ON 70 PRINT HEX$(HI+1,HI+1);
HEX$(LOW+1,LOW+1);
JD 80 PRINT ", DECIMAL ";KEY
CODE
AE 90 GOTO 20
```

Program 3: Screen Codes

```
OC 10 GRAPHICS 0
DP 30 SCREEN=PEEK(88)+256*PE
EK(89)
BI 40 REM FIRST: SCREEN COD
E ORDER
EI 50 FOR C=0 TO 255:POKE SC
REEN+C,C
OI 60 NEXT C
CO 70 REM THEN: (3 SPACES)ATA
SCII ORDER
HA 80 SCR2=SCREEN+40*8
DH 90 FOR C=0 TO 255:CHAR=C
HP 100 IF C>127 THEN CHAR=C-
127
EK 110 IF CHAR<32 THEN CHAR=
C+64:GOTO 140
NB 120 IF CHAR>95 THEN CHAR=
C:GOTO 140
ME 130 CHAR=C-32
JB 140 POKE SCR2+C,CHAR
BI 150 NEXT C
DH 999 GOTO 999:REM WAIT FOR
BREAK KEY
```



INSIGHT: Atari

Bill Wilkinson

Binary Files, Unite!

I've had several people write me that various programs designed for use with binary (machine language) files don't work with Atari's *Macro Assembler* (AMAC), OSS's *MAC/65*, or a couple of other assemblers. Or possibly a program will work with a small binary file produced by these assemblers, but not with a larger one. Why all these problems when the simple Atari *Assembler/Editor* cartridge works so well?

The root of the problem is the Atari Disk Operating System definition of a binary file, so let's examine that first. (Besides, maybe we'll learn a few extra goodies on the way.) A legal Atari binary file has the following format:

1. A header of two bytes, each with a value of 255 (hex \$FF).
2. Two more bytes indicating the starting address of a *segment* of the binary file. The two bytes are in standard 6502 low-byte/high-byte order.
3. Two more bytes indicating the ending address of that same file segment.
4. A sequence of bytes which constitute the actual binary code to be loaded into memory for the segment defined by the preceding four bytes. The number of bytes may be determined by subtracting the starting address from the ending address and then adding one.
5. If there are no further segments, there should be no more bytes in the file.
6. If there are more segments, then repeat this sequence of steps starting at either step 1 or step 2.

And that's it. A really neat, clean, format. Watch out for that last step, though. First, it says that the number of segments is theoretically unlimited. Second, it says that header bytes (dual hex \$FF bytes) may occur at the start of any segment. It also implies that there is no

particular order necessary to a binary file; it's perfectly OK to load the segment(s) at higher memory addresses before the one(s) at lower addresses.

RUN And INIT Vectors

Before moving on, there are two other niceties about DOS binary files worth knowing. When DOS loads a binary file (including an *AUTORUN.SYS* file at powerup), it monitors two locations. The simpler of the two is the RUN vector. Before DOS begins loading the binary file, it puts a known value into the two bytes at locations 736-737 (hex \$2E0-\$2E1). When the file is completely loaded (i.e., when DOS encounters the end of the file, step 5 above), if the contents of location 736 have been changed, then DOS assumes the new contents specify the address of the beginning of the program just loaded. DOS then calls the program (via a JSR) at that address.

The second monitored location is the INIT vector at address 738 (hex \$2E2). This vector works much the same as the RUN vector, but DOS initializes and checks it for *each segment* as the segments are loaded. If the INIT vector's contents are altered, then DOS assumes the user program wants to stop the loading process for a moment, long enough to call a subroutine. So DOS calls (via a JSR) at the requested address, expecting that the subroutine will return so the loading process can continue. This is a very handy feature. Most of you have probably seen it at work, such as when you run (or boot) a program which puts up an introductory screen (maybe just a title and a PLEASE WAIT message) and then continues to load.

The other important difference between the RUN and INIT vectors is that DOS leaves channel number one open while the INIT routine is

called. (DOS always opens and loads the binary file via this channel.) I suppose a really tricky program could close channel one, open up a different binary file, and then return to DOS. DOS would proceed to load the new file as if it were continuing the load of the original one. Most of the time, though, INIT routines should not touch channel one.

More On Segmented Files

Back to the main subject: Why do some programs have problems with binary files produced by some assemblers? Well, if all programs followed the complete binary file format as given by steps 1 through 6 above, there would probably be no incompatibilities. Unfortunately, many people who have used no assembler except the old cartridge have ignored segmented files. They have assumed that a binary file consists of steps 1 through 4, one time only, with a single large segment. Perhaps this is because many programmers first worked with Apple DOS, CP/M, and other operating systems with not-so-intelligent binary file formats. Or perhaps it is because the supposedly simple assembler cartridge is, in some ways, smarter than more advanced assemblers. In particular, the assembler cartridge will *not* produce multiple segments unless the programmer specifically asks for them (via an **=* directive to force a change to the location counter).

Yet other assemblers (including AMAC and MAC/65) never produce a segment longer than a particular size (usually a page—256 bytes—or less). If the programmer coded a longer segment, these assemblers automatically break it up into smaller pieces. Why? Probably to gain speed and lessen the work of assembly, since the assembler cartridge is doing a lot of work remembering the ending addresses

April 86

of segments.

Now, if my only concern were those few programs which don't properly load all binary files, I would simply have showed their authors the way to fix them. But there is a secondary advantage to programs which consist of larger segments: They load faster! Sometimes *much* faster. So this month I give you the BASIC program below, which takes any binary file and attempts to "unify" it. In particular, if the start address of one segment directly follows the end address of the preceding segment, they are consolidated into a single segment. And so on, so far as the space in BUF\$ allows.

And, last but not least, there's another minor bonus. Often, someone who writes an assembly language program purposely leaves space to be filled in later (e.g., by a filename, counter, etc.). If this reserved space occurs in the midst of code (probably not good practice, but it happens), it forces even the assembler cartridge to break the file into segments. But if the reserved space is significantly less than a sector (say under 50 bytes or so), it may be faster to let DOS load filler bytes. So you can change the value of the variable FILL in line 1160 (to 40, perhaps), and this program will automatically generate up to the specified number of fill bytes in an effort to better unify the file.

Whew! Was this month's topic too heavy for you? Then write me (P.O. Box 710352, San Jose, CA 95071-0352) with *your* suggestions

for a topic. No treatises please. One or two pages works best. Thanks.

Binary File Unifier

For instructions on entering this listing, please refer to "COMPUTE!'s Guide to Typing in Programs" in this issue of COMPUTE!.

```

GG 1110 REM allocate buffer
KI 1120 REM
DI 1130 BUFSIZE=FRE(0)-300
AK 1140 DIM BUF$(BUFSIZE)
II 1150 DIM FILEOLD$(40), FILENEW$(40)
KH 1200 REM
CJ 1210 REM get file name
KJ 1220 REM
NO 1230 PRINT "I need two file names: An existing"
EA 1240 PRINT " object file and a new file which"
EE 1250 PRINT " will get the 'unified' object code."
FG 1260 PRINT
AA 1270 PRINT "Existing file?"
DE 1280 INPUT #16, FILEOLD$
OB 1290 PRINT "(5 SPACES) New file?"
DI 1300 INPUT #16, FILENEW$
KJ 1400 REM
JC 1410 REM open files, validate existing one
KL 1420 REM
FJ 1430 OPEN #1, 4, 0, FILEOLD$
JD 1440 GET #1, SEGLOW: GET #1, SEGHIGH
KD 1450 IF SEGLOW=255 AND SEGHIGH=255 THEN 1500
PI 1460 PRINT :PRINT "Existing file: invalid format"
KD 1470 END
DF 1480 REM input file okay
LC 1490 REM
GH 1500 OPEN #2, 8, 0, FILENEW$
MF 1510 PUT #2, SEGLOW: PUT #2, SEGHIGH
KL 1600 REM
NO 1610 REM process a new origin
KN 1620 REM
AK 1630 BUFPTR=0

```

```

OO 1640 BUF$=CHR$(0):BUF$(BUFSIZE)=CHR$(0)
HB 1650 BUF$(2)=BUF$:REM zap buffer
ML 1660 PUT #2, SEGLOW: PUT #2, SEGHIGH
KN 1700 REM
AA 1710 REM process a segment
KD 1720 REM
IF 1730 GET #1, ENDLow: GET #1, ENdHIGH
BH 1740 SEGSTART=SEGLOW+256*SEGHIGH: SEGEND=ENdLOW+256*ENdHIGH
HE 1750 SEGLE=SEGEND-SEGSTART+1
HF 1760 REM read segment into buffer
HL 1770 FOR PTR=1 TO SEGLE
KH 1780 GET #1, BYTE: BUF$(BUFPTR+PTR)=CHR$(BYTE)
AG 1790 NEXT PTR
KN 1800 REM
MF 1810 REM check head of next segment
KP 1820 REM
JG 1830 GET #1, SEGLOW: GET #1, SEGHIGH
KK 1840 IF SEGLOW=255 AND SEGHIGH=255 THEN GET #1, SEGLOW: GET #1, SEGHIGH
OL 1850 SEGNEXT=SEGLOW+256*SEGHIGH
ED 1860 GAP=SEGNEXT-SEGEND-1
HC 1870 IF GAP>FILL OR GAP<0 THEN 2000
KA 1880 BUFPTR=BUFPTR+SEGLE+GAP
ED 1890 IF BUFPTR+256>BUFSIZE THEN 2000
ML 1900 GOTO 1700
KG 2000 REM
DJ 2010 REM need to dump buffer to
LA 2020 REM prepare for new origin
KJ 2030 REM
LE 2040 PUT #2, ENdLOW: PUT #2, ENdHIGH
OG 2050 FOR PTR=1 TO LEN(BUF$)
LC 2060 PUT #2, ASC(BUF$(PTR))
PD 2070 NEXT PTR

```

Write computer programs at home...

BE SUCCESSFUL WITH COMPUTERS

Self-paced, one-on-one instruction



Unique study program includes word-processing, bookkeeping, data base management, mailing lists, inventories, and all about personal computers.

No experience needed! Learn *once* before you decide on a computer.

Professional, home and business applications.

HALIX INSTITUTE
CENTER FOR COMPUTER EDUCATION DEPT. 614
1543 W. OLYMPIC, #226
LOS ANGELES, CA 90015-3894

YES! Send me free information on how I can learn about computers and programming at home!

Name _____
Address _____
City _____ State/Zip _____

PREMIUM QUALITY! LIFETIME WARRANTY!

DISK SALE!

Made by top USA makers, not low-end or seconds. We buy truckloads of major makers' overproduction. We can't print the maker, but you'll recognize them as among the HIGHEST QUALITY PREMIUM DISKS MADE. Certified, guaranteed 100% error free. MONEY BACK SATISFACTION GUARANTEE!

from **33¢**

Prices are per disk	100	500	1K	5K	10K	10	100	500	1K	5K
SS/DD 35mm for Apple II series +UDA	45¢	42¢	39¢	36¢	33¢	58¢	53¢	50¢	47¢	44¢
SS/DD for single side computers +UD1	59¢	57¢	55¢	50¢	46¢	75¢	67¢	65¢	63¢	57¢
DS/DD for IBM PC & compatibles +UD2	67¢	60¢	56¢	56¢	54¢	84¢	75¢	68¢	66¢	64¢
DS/QUAD DENSITY (80psi) +UD4	1.09	1.04	89¢	82¢	89¢	1.10	1.11	1.05	1.02	99¢
DS/HD "HI DENSITY" for IBM AT +UAT	2.15	2.12	2.10	2.06	1.99	2.40	2.25	2.22	2.20	2.18
3 1/2" single side +UD3	1.96	1.93	1.90	1.87	1.84	1.99	1.96	1.93	1.90	1.87
3 1/2" double side +UD6	2.85	2.85	2.45	2.40	2.30	2.80	2.70	2.55	2.45	2.40
8" SS/DD self sect. +UDH						1.20	1.00	1.00	1.00	1.00
5 1/4" SLEEVES #24 white wavy +SLV	4¢	3.5¢	3¢	2.8¢	2.6¢					
5 1/4" SLEEVES white ivory +ITV	6¢	5¢	4.5¢	4.3¢	4.1¢					

INCREDIBLE PRICES ON COMPUTER ACCESSORIES!
UNFILE-100 holds 100 disks, with lock & key, removable top. \$13.88
UNFILE-70 holds 70 disks, like "Tip-o-file." #HLE70 \$8.88
UNFILE-10 LIBRARY CASES, holds 10 disks, clear plastic. #FILE10 99¢
UNIPAK 5 1/4" DISK MAILERS, rigid cardstock, up to 3 disks HUNIPAK 10 for \$5
LETTER QUAL COMPUTER PAPER, 2500 sheets, 20lb microport 8 1/2x11" #LQP \$29
DRAFT QUAL COMPUTER PAPER, 3300 sheets, 15lb std-port 8 1/2x11" #DQP \$29

HOW TO ORDER: Pay by MC/VISA/Amex/COD, or send check with order, minimum order \$20, add \$3 for shipping (call for shipping rates on paper). We can ship upon account to schools with good credit, minimum purchase order \$100. FOB Unitech. All orders must include daytime phone and STREET address. Money-back 30 day satisfaction guarantee. Send for FREE CATALOG listing 100's of discount computer supplies!

TOLL FREE (800)343-0472
UNITECH IN MASS: (617) "UNI-TECH"
200 HURLEY ST.
CAMBRIDGE, MA 02141

ATTENTION T.I. 99/4A OWNERS

- Diskettes - 59¢ each! Your choice SS or DD
- 512K Now Available for the 99/4A!
- 99/8 Level 4 Computer Upgrade Now Available
- Over 1500 Hardware and Software Accessories at Similar Savings

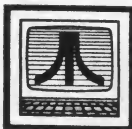
THE WORLD'S LARGEST COMPUTER ASSISTANCE GROUP

Now serving over 35,000 members worldwide with the best in technical assistance, service, and products for the Texas Instrument 99/4A Computer.

To become a member and receive newsletters, catalog, technical assistance and membership package, send \$10.00 for a ONE Year Membership ship to:

99/4A National Assistance Group
National Headquarters
P.O. Box 290812
Ft. Lauderdale, Florida 33329
Attention Membership Division
For Further Information Call 24 Hours
(305) 583-0467

April '86



Atari Printer Trivia

This month's COMPUTE! is a printer issue, so I decided to break with (my) tradition and write a column on printers. Before we start, though, an erratum: My April column listed a program designed to "unify" a machine language file on disk. But when I sent the column to COMPUTE!, I accidentally included a couple of older versions of the program on the same disk. Guess which version got published? Anyway, COMPUTE! listed a corrected version in the article entitled "Custom Characters for Atari SpeedScript" in the May issue. (By coincidence, it happens that my program is needed to unify the COMPUTE! DISK version of SpeedScript before installing a custom character set.) On to the printers.

Number, Please

John Skruch at Atari gets credit for revealing this first tidbit. You are all aware that disk drives can be assigned device numbers (from D1: to D8:, though Atari drives can only go to D4:), but did you know that printers can have numbers, also? If you have an 800XL, 65XE, or 130XE, you may connect two or more printers at the same time and direct output to one or the other. From BASIC, for example, it's as simple as typing

LIST "P2:"

or

LIST "P5:"

Two major drawbacks: all printers still respond as P1:, so using P1: or just P: when two printers are attached leads to humorous and/or disastrous results. Since many programs always address printers as P:, this trick may be useful only in your own programs. Also, only the following printers have these secondary numbers:

Printer	Secondary Number
850	P2:
1025	P3:

1020 P4:
1027 P5:
1029 P6:

(The 850 refers to *any* printer attached via an 850 Interface Module. The 1029 printer is rare in the U.S.)

The fact that the 850 can handle different printer numbers indicates that provision for this feature was included as far back as 1979 (when the 850 was first made). Do you wonder why nothing was said sooner? Why don't the 400, 800, and 1200XL work with multiple printers? Do any other interface modules (from third-party vendors) have secondary device numbers? A prize for the best answer.

The Nine-Minute Nap

If you have a 1027 printer which is not lucky enough to be hooked up to an XE computer, you've probably experienced the infamous sleeping printer bug. Sometimes the 1027 just suddenly stops printing. Many people believe they need to reboot their system to wake the printer up. Actually, after about nine minutes, the printer just as suddenly springs to life again. The reasons for this are too strange and lengthy to go into here. Suffice it to say that the problem has existed since the first Atari computer was built and is related to the (also infamous) sleeping disk drive phenomenon—though the drive only sleeps about five seconds. You'll be pleased to know that Atari's newest operating system ROMs in the XE computers finally fix the problem.

If you do have a 1027, but don't have an XE, and still want to fix this problem, type in, save, and run the accompanying program. It automatically seeks out the LOMEM value for your system and then creates an AUTORUN.SYS file to patch the timeout problem. The AUTORUN.SYS file will load at that LOMEM point and then

move LOMEM above itself. Since it reads the current LOMEM, be sure to create the AUTORUN.SYS file on the same disk, *booted in the same fashion*, that you later want to use. This means, for example, that any special drivers (RAM disk, RS-232, and so on) must be installed before you run this BASIC program.

For a more specific example, let's say you intend to use the 850's R: driver with AtariWriter and the 1027. You must start by booting the 850's AUTORUN.SYS file to install the R: driver in memory, *then* run the program below.

Also, if you have true double-density drives (not "enhanced density" 1050s), boot with double-density disks inserted. This patch should work with almost any DOS, such as DOS XL, SpartaDOS, DOS 2.5, or whatever—but I wasn't able to test them all.

Two final points: If an AUTORUN.SYS file already exists on the disk when this program is run, the 1027 patch is appended to that file. Again, using the 850 as an example, this means you'll have a single file which serves two purposes: It boots the R: driver and makes the 1027 patch. Finally, line 170 of the listing is a REMark; if you delete the REM to enable this line, it reserves two pages (512 bytes) of extra memory. If you have any trouble running this patch, try deleting the REM. For instance, if your system has more than one disk drive, you might want to make this change.

Obviously, I did not develop this program by arbitrarily typing in funny numbers for my DATA statements. I started with a program written by Joe Miller (formerly of Atari), then fixed it so that it survives SYSTEM RESET, is relocatable, moves LOMEM if appropriate, and does not install itself twice. If you're interested in studying the source code for this

June 86



INSIGHT: ST

Bill Wilkinson

Exploring The ST

Hi—welcome to “Insight: ST,” *COMPUTE!*’s new monthly column for the Atari ST-series computers. Over the coming months, we’re going to help you become more familiar with the ins and outs of the Atari ST, its operating system, GEM, and ST BASIC. The ST series is the most powerful line of computers ever released by Atari—one of the most powerful in the industry, in fact—so there’s a great deal to learn and explore.

Before getting started, I want to reassure those of you who still own and use the eight-bit Atari 400/800, XL, and XE computers. Veteran readers will recognize that I’ve been writing the “Insight: Atari” column on these machines for the past five years in *COMPUTE!*. This new column *does not* mean that we’re dropping Insight: Atari. In fact, I plan to continue writing Insight: Atari in addition to Insight: ST for the next few months. Eventually I’ll turn over Insight: ST to someone more specialized in ST BASIC. Don’t be surprised, though, to find occasional ST tidbits in Insight: Atari as well. Both columns will be of continuing interest to all Atari enthusiasts.

The ST In Perspective

Just what *is* an Atari ST computer? Even if you already own an ST, I may have some surprising answers for you.

From a hardware viewpoint, the ST is most commonly compared to the Apple Macintosh and the Commodore Amiga. Indeed, it shares characteristics with both. All three use a Motorola 68000-series microprocessor, 3½-inch floppy disk drives, bit-mapped screen display, a range of peripheral interfaces, and more—generally the things we’ve come to expect from today’s advanced personal computers. Both the ST and the Amiga have one advantage over the Mac-

intosh: color graphics (though admittedly only the Amiga uses a sophisticated graphics processor chip to display true sprites).

Even the user interfaces of the three machines are similar: All have a system of icons, multiple screen windows, and a mouse controller to visually display and manipulate the contents of disks and perform general “maintenance” chores.

Finally, as long as we’re making comparisons, we should be fair and mention that the Macintosh has much, much more software available for it than either of its competitors. But that situation is changing rapidly, even as I write.

What makes the ST stand out from other computers? Well, the Atari marketing department has a whole series of answers, but let me tell you the ones which impress me. First and foremost is the built-in hard disk port. It’s capable of transferring data to or from a hard disk (or a network or external RAM disk or whatever) at a rate of up to *1,300,000 bytes per second*. That means you could, in theory, fill the 512K RAM memory of a 520ST in under half a second.

The Allure Of Speed

Theory is nice, but what does this mean in practice? Well, for me (or any other programmer) it means that after writing source code with a text editor, I can save the source file to disk, exit the editor, load a compiler, compile the source code I just saved, link the resulting object code with both system and GEM libraries, and maybe (if I swallow fast) finish eating a bite or two of a sandwich. Elapsed time: between 10 and 15 seconds, depending on the size of the program. On an IBM PC with a hard disk, it would take four to six times as long. And on the Macintosh, most external hard disks aren’t much faster than the ST’s floppies.

Thanks to its fast processor and amazing hard disk speed, for sheer computing power there is probably no “home” computer available which can touch the ST. Exception: If you’re doing heavy work with floating-point math (for example, scientific or engineering computing), an IBM PC with 8087 floating-point chip will win hands down. (Are you listening, Atari?)

The only other hardware features which are distinctively Atari are the built-in MIDI (Musical Instrument Digital Interface) ports, the cartridge slot, and the absolutely beautiful *black-and-white* display. Now MIDI will be of interest to musicians, and the cartridge port may have some interesting future applications (perhaps a way to get that fast floating-point chip?), but the surprise here is the 640 × 400 monochrome display. Why do I rave about this on a machine with advanced color graphics?

Although I enjoy color displays, I will probably never create one. I’m not particularly artistic and I don’t write games. But I *do* write programs. Which means I appreciate an easy to read, rock-steady display. Atari went to the trouble to equip the ST with a completely separate monochrome video port, and its quality is nothing short of amazing. And, besides, it costs \$200 less than a color system. (But be forewarned: Many games only run on a color monitor. Poor software design, in my opinion, but that’s how it is.)

Next month, we’ll begin turning the Atari ST inside out and exploring the intricacies of TOS, its multilevel operating system. **C**



INSIGHT: ST

Bill Wilkinson

ST System Software, Inside Out

Okay, you've got your shiny new ST computer plugged in and running. You can use the mouse to select programs, copy files, and format disks. It's fun, and it certainly is easier to learn than figuring out what

COPY B: \SYSTEM \MSG5.TXT/A=A: SPCL*.MS?

is supposed to mean. (That's a real and possible IBM PC command.) But how did this system get built? Glad you asked.

Collectively, the software built into the Atari is called TOS (Tramiel Operating System). When the 520ST was first shipped, TOS was delivered on a disk. If you're still using the disk-based TOS, stop now. Go out and buy the ROM (Read Only Memory) version of TOS. It should cost no more than \$25 or so. Installation is not too difficult, though if you have as many left thumbs as I do, you might be advised to find a dealer or service center to install the chips for you (maybe \$20 to \$30 extra).

TOS in ROM is actually composed of six separate pieces. Usually, we lump these pieces into two groups of three: the graphics processing section and the underlying operating system. As we shall see, that operating system—a derivative of CP/M-68K—is very similar to MS-DOS and PC-DOS, which are both derivatives of CP/M.

BIOS, XBIOS, And GEMDOS

In one sense, we can say that the lowest level of the ST's operating system is the BIOS (Basic Input/Output System), a holdover from the earliest days of CP/M. At this level, we find routines for such basic tasks as sending a single character to a device, reading a disk sector (by sector number—a very dangerous practice), and so on. In CP/M, there was only one legitimate reason to call the BIOS directly: speed. With TOS, though, only

the BIOS provides some of the facilities which even a moderately sophisticated program will need (admittedly, often because of bugs in the upper levels of the operating system).

On the ST, a BIOS call is implemented as a TRAP instruction in 68000 machine language. All the necessary parameters, including the BIOS call number, are passed onto the stack. If you aren't quite sure what we're talking about, don't worry about it. Virtually every programming language for the ST has some way to use these routines which mask the mechanics of TOS calls. It's a good thing, too, since some of those mechanics can get pretty hairy.

The next higher component of TOS is the XBIOS (eXtended BIOS). XBIOS supplies the Atari-unique routines needed to do such things as access the sound registers, screen hardware, and so on.

The third component of the operating system is called GEMDOS (Graphics Environment Manager/Disk Operating System). Actually, this is a misnomer. The GEMDOS routines have nothing whatsoever to do with graphics. GEMDOS is essentially an MS-DOS or PC-DOS emulator. Want to open a file? Read a block of bytes? Get a character from the keyboard? Given the differences between the 68000 of the ST and the 8088 of the IBM PC, the similarities between GEMDOS calls and MS-DOS calls are almost scary.

GEM, VDI, And AES

Okay, enough about the underlying operating system. Let's take a look at the graphics systems which comprise GEM. The most familiar part is the GEM desktop which appears when you turn on your ST. But the desktop is not really a special program at all; it simply calls the lower-level routines. Again, there are three levels of graphics routines.

The lowest-level graphics, not officially part of GEM but merely one means of implementing it, are those called the *Line-A Routines*. This sounds cryptic, but it simply refers to the fact that certain machine instructions of the 68000 (including those of the form \$Axxx hex—hence "line-A") are reserved and cause a special hardware trap into the OS. As you might expect, routines implemented in this fashion are of the most fundamental type: draw a line, plot a point, and so forth. Most are very fast.

The next level up in graphics is the VDI (Virtual Device Interface). In theory, VDI is capable of supporting several types of graphics devices in a uniform fashion. For example, you might use the same set of calls to draw a curve on a plotter or on the screen. Unfortunately, no such drivers are yet available (or, as far as I can tell, even in the works) for the ST. Still, the possibility exists.

VDI does all the actual graphics work on the ST. It draws simple rectangles, bordered ovals, and text in various styles, sizes, and colors. Someone who learns nothing on the ST except how to call VDI could still do some remarkable graphics work.

Finally, at the highest level, is AES (Applications Environment System). AES is what GEM uses to present you with that nice, pretty desktop, complete with menus, dialog boxes, alert boxes, windows, and icons. Perhaps more important to programmers, though, is the fact that AES allows us to use all the features of GEM in a relatively consistent, properly desktop-compatible manner. It is through this mechanism that even a lowly spreadsheet program can have drop-down menus, mouse-controlled windows, and all the rest of those impressive features. C

From The Editor's of **COMPUTE!** Magazine and
Optimized Systems Software, Inc.

INSIDE ATARI[®] DOS

Compiled by Bill Wilkinson,
Optimized Systems Software, Inc.

Published by **COMPUTE! Books**,
A Division of Small System Services, Inc.,
Greensboro, North Carolina

ATARI is a registered trademark of Atari, Inc.

A
Small System
Services, Inc.
Publication

Introduction

BEING A HISTORY OF TWO BIRTHS “COLEEN” AND “CANDY”

I don't know exactly when the concept of the Atari Computer was developed within the corporate mind of Atari, Inc., nor do I know all of the people responsible for nursing that concept into reality. The following history covers the relationship with Atari, Inc., during the evolution of the system software.

Sometime in early 1978, when the Atari 800 and 400 were still called “Coleen” and “Candy” and were still in the breadboard stages, Atari bought a copy of the source for Microsoft 8K BASIC. This version of BASIC was fundamentally the same product that was implemented by Commodore in the early PETs, was used by OSI, and was a close ancestor of Applesoft. Six months and many, many Atari man-hours later, that 8K BASIC was *almost* functioning properly on the Atari prototypes. But buying source for a program buys you just that: source. Generally, you also receive little documentation, sometimes obscure code, no guide to modification, and no real support. What to do? The products were due to be shown in early January, 1979, at the Consumer Electronics Show (CES) in Las Vegas, Nevada.

Enter Shephardson Microsystems, Inc. (SMI), my employer at that time. Though little known by the microcomputer public, SMI had already produced some very successful, private labeled microcomputer software. Among our better-known efforts were the original Apple DOS, Cromemco 16K Extended BASIC, and

Cromemco 32K Structured BASIC (just being completed at that time). Also, we had done some work for Atari on a custom game processor. (Which used a 12-bit ROM and 5-bit RAM configuration and was well received at Atari, but never produced.)

Coincidentally, about that same time SMI had also purchased source for Microsoft 6502 BASIC. After producing Apple's DOS, we had the bright idea of mating the Apple II peripheral bus with the KIM/SYM/AIM system bus (and it still seems like a good idea to us, but ...). The idea was to provide a disk system (Apple's) to the Single Board Computer market. Needing a BASIC to sell with the system, we plunked down a few grand and purchased Microsoft's. Though it looked to us like it would be difficult to modify, we were intending to resell it with a minimum of changes, so it seemed appropriate.

A New BASIC?

Re-enter Atari, some time in the late summer of 1978, asking if SMI could help them. With Microsoft BASIC? Well ... we really didn't want to, but ... Could we propose a new BASIC? We talked. And had meetings, and a study contract, and more meetings, and finally we wrote a specification for a 10K, ROM-based BASIC. (I still have a copy of that spec, and it's amazing how little the final version deviated from that original.)

Of course, in the middle of all these discussions, Atari naturally divulged how their (truly superb) ROM-based Operating System would interface both with BASIC and with various devices.

Somewhere in here, my memory of the sequence of events and discussions becomes a little unclear, but suffice it to say that we found ourselves making a bid on producing not only a BASIC for Atari, but also the File Manager (disk device driver) which would change Atari OS to Atari DOS.

Sometime in late September, 1978, the final proposal was made to Atari, and it was accepted by them shortly thereafter. In mid-October, 1978, we received the go-ahead. The project leader was Paul Laughton, author of Apple DOS. The bulk of the work ended up being done by Paul and Kathleen O'Brien. Though I was still involved in the finishing touches on Cromemco BASIC, I take credit for designing the floating point scheme used in Atari BASIC. Paul Krasno implemented the math library routines following guidelines supplied to us by Fred Ruckdeschel (author of the acclaimed text, *BASIC Scientific Subroutines*). And, of course, much credit must go to Mike Peters, our combination keypuncher/computer operator/junior programmer/troubleshooter.

Since we obviously couldn't have the Atari machines to work on (they hadn't been built yet), the first step was to bring up an emulator for Atari's CIO ("Central Input-Output," the true heart of Atari's OS) on our Apple II systems. With Paul Laughton leading the way (and doing a lion's share of the work), the pieces fell together quickly. "Little" things had to be overcome: the cross-assembler was modified to handle the syntax table pseudo-ops, the 256-byte Apple disk sectors had to be made to look like 128-byte Atari sectors, the BASIC interpreter seemed to function, but was waiting for the floating point routines. And there are funny things to tell of, also. Like our cross-assembler, running on an IMP-16P (a 1973 vintage, 16-bit, bit-sliced PMOS microprocessor) that used keypunched cards for input, a floppy disk (with no DOS) as temporary storage, and a paper tape punch as output.

Somehow, Kathleen and Paul guided the two programs unerringly toward completion. On December 28, 1978, Atari's purchasing department at last delivered a signed copy of the final purchase order. It called for delivery of both products by April 6, 1979. There was a clause which provided for a \$1,000 per week incentive (if we finished early) and penalty (if we finished late). What is especially humorous about that December 28th date is that the first working versions of both BASIC and FMS had *already* been delivered to Atari over a week before! That is *fast* work.

Fortunately, then, Atari took their new Atari BASIC to CES. Unfortunately, there was a limit on the amount of incentive money collectible. Oh, well.

In the months that followed, SMI fixed bugs, proofread manuals, and worked on other projects (including the Atari Assembler/Editor, which was mostly Kathleen's effort). The nastiest bugs in BASIC were fixed by December, 1979, but it was too late: Atari had already ordered tens of thousands of BASIC ROMs. The FMS bugs were easier to get fixed, since DOS is distributed on disk.

In mid-1980, Paul Laughton once again tore into FMS. This time, he modified it to handle the ill-fated 815 double-density disk drive and added "burst I/O" (and there will be much more about both these subjects in the technical discussion that follows).

In late 1980, and early 1981, Bob Shepardson, owner of Shepardson Microsystems, Inc., decided that the pain and trouble of having employees wasn't justified by the amount of extra income (if any) that he derived. Though we still occasionally function in a loose, cooperative arrangement, the halcyon days of SMI seem to be over.

A New Beginning

I negotiated with Bob Shepardson for his rights to the Atari products (FMS, BASIC, and the Assembler/Editor) and their Apple II counterparts. Thankfully, Atari had purchased from SMI only a non-exclusive right to distribute these products. SMI had retained the rights to license other users on a similar non-exclusive basis (and, indeed, SMI sold a version for the Apple II during most of 1980).

So now it was frantic time again: this was February 25, 1981, and the West Coast Computer Faire was April 3rd. But our brand new company, Optimized Systems Software, arrived on time, bringing with it BASIC A+, OS/A+ and EASMD. All three were enhanced, disk-based versions of the original Atari programs (and, in fact, derived some of their enhancements from the previous OSS Apple II products).

The products have been well received by the Atari user community, in part due to the fact that they are truly compatible, yet enhanced, versions of standard Atari software.

Why This Book?

The decision to publish these listings was not an easy one to make; and it is, in its own way, an historic occasion. After all, have you ever seen anyone offering source or listings of CP/M, the most popular of all computer operating systems? Since Atari, to their credit, has honored the original agreement with SMI and not released either source or listings without permission, the responsibility for doing so seemed to rest with OSS.

But Atari has set a powerful precedent by publishing the listings of DUP (their portion of DOS 2.0S) and the OS ROMs. The clamor from Atari users for the source for FMS finally even reached us, so we have bowed to the inevitable, and honored the same commitment that Atari has made: to release as much information and aid as possible to the user community.

We hope that the users will appreciate these efforts and, in turn, respect our rights and Copyrights. As long as there is a mutual respect and benefit, you, the user, can expect continued support.

About This Book

With the release of this book, the dedicated Atari enthusiast can examine all the inner workings of Atari DOS and modify his (or her) system to his heart's delight. Rather than simply publish listings, we have chosen also to provide a complete guide to the workings of FMS. Although the listing itself is relatively clear and commented, all

but the most expert would have trouble plowing through some of the tortuous logic necessary in such a program. The guide included here describes all aspects of the FMS, including the external view, the charts and tables, the various interfaces, and (in copious detail) the functions of the individual subroutines (including complete entry and exit parameters).

There is much of value here even for the person who never intends to modify Atari DOS. We feel that FMS is a fairly well-structured, relatively sophisticated, system level assembly language program. We hope that most users will gain by the insights presented here.

We would welcome any notes you would care to send pointing out errors either in the DOS or in this book.

Bill Wilkinson
Optimized Systems Software
Cupertino, California
February, 1982

Chapter One

ATARI DOS OVERVIEW

The standard Atari Disk Operating System, DOS 2.0S, consists of four separate elements, ranked as follows in order of their "visibility" to the average DOS user.

1. DUP - Disk Utility Package
2. CIO - Central Input/Output
3. FMS - File Management System
4. SIO - Serial Input/Output

It is helpful to understand the entire Input/Output (I/O) process. While this book is intended to give detailed information on the workings of FMS, this overview will attempt to at least show how the four elements of DOS are connected. To this end, we would first call your attention to Figure 1. This figure is, itself, an overview of the entire Atari I/O system, including indications as to how and where data and control flows between the various elements thereof. Figures 1-1 through 1-4 show "close-ups" of portions of this diagram as they relate to the four elements of DOS.

In these figures, the rectangular boxes represent system elements, and are appropriately labeled. The wide, lettered arrows represent the flow of data (via buffers, control blocks, or even registers) between the various elements. The narrow, numbered arrows show how and where control, and control information, is transferred.

4-1. Disk Utility Package

DUP (which shows as "DUP.SYS" in a disk directory listing) is the most obvious and visible element of Atari DOS. DUP's function is to provide the user with keyboard access to the various file management functions in FMS. It does so via the menu which is displayed when, for example, the user keys "DOS" from BASIC. Actually, the menu offers several options which are not directly a part of the FMS (e.g., copy and duplicate files). Refer to the Atari Disk Operating System II

Event Management and Windows in Pascal

Program by Mark Rose
Text by Bill Wilkinson

Event management and windows in the GEM environment are two of the most misunderstood topics of ST programming. This article sheds some light on these areas and shows you what a well-structured GEM program looks like.

When Apple introduced the Macintosh computer, the company dubbed it "the computer for the rest of us." Since a fully equipped Macintosh originally cost as much as a pretty fair used car, Apple certainly wasn't referring to the price. Instead, the emphasis was that the Macintosh is a powerful machine, easy for even a complete novice to use. So true. Imagine being able to copy a file by pointing to a little picture and then dragging an outline of that picture to another picture which shows a second disk drive. No more cryptic commands such as

PIP/QVF B: \WORK \ = A: \BUSY *.DAT

or worse. Apple's question: What could be better?

Atari's response: How about a computer that is affordable for the "rest of us"? Seen at first glance, the desktop of the Atari ST looks amazingly like that of the Macintosh. Upon closer examination, you see several not-so-subtle differences. Just as an example, the Macintosh allows custom pictures (icons) to be drawn and associated with particular types of files. The Atari ST has only three fixed icons—folders, programs, and data files—which are often more than a little confusing because, for example, BASIC programs are considered to be data files.

Still, the most outstanding features of the Macintosh are maintained: a mouse that is easy to use, a visual desktop, and, naturally, windows. The logical consequence of all this is that programmers who want their products to mesh well with the

ease and grace of the ST must learn how to "do windows."

This discussion then will focus on two of the most misunderstood topics of ST programming: event management and windows in the GEM environment. The program accompanying this article is designed to show a well-structured GEM program. Since we are principals in Optimized Systems Software (OSS), and since OSS produces a popular language for the Atari ST known as *Personal Pascal*, it is not surprising that this program is written in *Personal Pascal*.

You say you don't have *Personal Pascal*? No problem. This isn't a practical program anyway (though it certainly could serve as the basis for one). Its purpose is simply to instruct you in programming techniques, and the methods and algorithms apply equally well, whether you program in Pascal, C, Modula-2, assembler, or any language which gives you full access to the power of GEM. (Which is another way of saying that BASIC and Logo users don't really need to know much of this stuff. But when you get tired of the self-imposed limitations of those languages, come back and read this again.)

Personal Pascal comes with a special library of functions and procedures that give programmers access to a reasonably good selection of GEM features. In reading the example program and the following material, you will find that several of these special library routines are not explained. This is only reasonable; who cannot deduce the purpose of a procedure named `Paint_Color`? On the other hand, we'll briefly explain routines with less obvious names and/or purposes. Long descriptions are unnecessary, since *Personal Pascal* users can find them in their manuals, and users of other languages will either have to invent their own routines or use names and calling sequences dissimilar to those given here. We have attempted to make the program as readable as possible so that it will not cause any confusion.

What Window_Demo Does

In good programming style, we've given our program a readable name: "Window_Demo." In typical computer style, when we put it on disk, we were forced to call the program "WINDDEMO", since TOS allows only eight characters in a filename. After you have compiled the source code (or otherwise made a copy of the runnable program), you simply need to click on the icon in the desktop which bears the name

"WINDDEMO.PRG". It will load and start to run.

The first obvious change will be in the menu bar at the top of the screen. Along with the familiar Desk designation, you will find Sizes, Shapes, and Patterns. If you move the mouse pointer up to the menu bar, you can touch on one of these names and be rewarded with a drop-down menu. For example, if you choose Sizes, a drop-down menu will appear containing the selections Small, Medium, and Large. Beside one of these selections will be a checkmark. If you choose a different selection and click on it, the drop-down menu will disappear, but when you reselect it, your new choice will have the checkmark next to it.

Similarly, there are three shapes (Square, Circle, and Wedge) and three patterns (Solid, Checkered, and Open); you can choose one shape and one pattern. The real magic of this program occurs, however, when you point the mouse at the Desk title in the menu bar: In addition to any desk accessories already present on your boot disk, a selection titled New Window appears. Choose it by clicking on it, and the action will begin.

First, a window appears, filling the desktop except for the menu bar area. Its title is Window 1 (or a higher number if this is the second or subsequent time you have chosen New Window). This window is typical of the characteristics of GEM windows in several ways: it has a title, move bar, close box, and size box. The implication is that you can point the mouse at the size box and drag the corner of the window to make it smaller. You can also move the window by dragging it via the bar containing its title, though we suggest you do this *after* you have shrunk it somewhat. If you can run the "WINDDEMO" program at this time, we suggest that you create several windows right away, shrinking and moving each to different sizes and positions on the screen.

Now, the meaning of Sizes, Shapes, and Patterns becomes obvious: If you point the mouse somewhere in the interior of the frontmost window, a shape of the type you "check-marked" will appear. Its size and interior pattern will also match your choices. If you have a color monitor, the program chooses three colors cyclically (more on this later). You can move the mouse pointer and click to request as many as ten shapes per window.

Notice that you can change your choice of shape, size,

and pattern at any time, either while working in the same window or before moving to another window. The best part, though, comes when you move or resize your windows again: All of your work is not forgotten. The objects remain in the same relative positions within the windows.

If you ran this program in a nonwindowed environment (for example, on a typical eight-bit computer like an Atari 130XE), none of this would be very difficult or extraordinary. But let's take a closer look at the programming techniques necessary to accomplish all this in GEM's multiwindow environment.

When the Bottom Is the Top

We'll follow the logical flow of the program, not the physical order of the listing. With Pascal programs, that usually means we have to start at the bottom of the listing, because the top of the program starts there.

Before we start, let's introduce some of the **types** and **variables** used in the program. In particular, you need to know that any **types** which seem undefined in the program are undoubtedly described in the file "GEMTYPE.PAS", which is included in this compilation via the following program line:

```
{ $I GEMTYPE.PAS }
```

The \$I is also used to include the files "GEMCONST.PAS" (predefined constants) and "GEMSUBS.PAS" (external support routines). This mechanism, or similar ones, is common to many compiled languages. Some languages, such as C, have a feature like this in the definition of the language. Others, such as Pascal, have acquired one through common usage: *Personal Pascal's* use of \$I has historical precedents.

In this program the most difficult **type** to understand is probably `window_info`, the last **type** defined. If you remember what appeared on the screen when we ran this program, you will soon see why this **type** is so complex: Each window has a title (name) and may contain from zero to ten objects (`obj_count`). The objects must be described, and for this we use an `object_list`.

In fact, this list is an array of ten elements, each of which is called an `object_info` and each of which describes the shape, location (both horizontal and vertical within the window), size, pattern, and color of a corresponding object. With a little perusal, you will see the rationale for the names and

and more —

detailed description of program

START

THE ST QUARTERLY

U.S.A. \$14.95 CANADA \$19.95

Disk Enclosed



TM

Winter 1986

Volume 1, Number 3

Buyer's Guide:

Choose From Over
300 ST Products

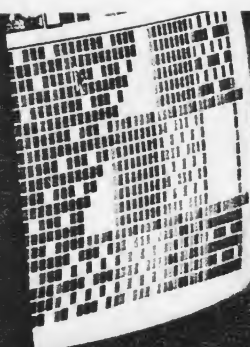
The START Interview: Jack Tramiel

Paint Like A Pro With DEGAS or NEO

Bill Wilkinson Rates ST Programming Languages

Hot Sounds Unleash the Power of Your Sound Chip

Personal Pascal Tutorial: Create an Address Book, Addresser and Labelmaker



Tower of Babel

Pick a Language, Any Language...

BY BILL WILKINSON

Bill Wilkinson, renowned Atari guru and computer language maven, opines over ADA, BASIC, COBAL, Forth, Fortran, assembly, and much much more. Looking for a language? Ask the man who speaks in tongues.

For those of us who have been accustomed to the Atari 8-bit world, the array of computer languages already available for the Atari ST computers is staggering. With promises of even more languages soon, how can you decide which one to use?

In this article I will tell you what I think are the advantages and disadvantages of some of the available ST languages. Note carefully that I said "I think." Almost every programmer has a favorite language, and thinks that every other language is either fascist, anarchistic, unstructured, strait-jacketed, backwards, sluggish, or stupid, depending on which language is being described by which fanatic.

When the editors of START asked me to write this article I hesitated. After all, I am associated with Optimized Systems Software (OSS), producers of one of the more prominent languages for the ST (Personal Pascal), and soon to offer perhaps the most esoteric language (Personal Prolog). I feared that any effort I might make would be viewed by some as a conflict of interest.

Another telling point—I have not used all the languages available, so how can I judge them? Truthfully, I can't. So some of what you read here will be the result of many, many conversations with other programmers or of my own experiences with similar languages on other computers.

Through all of this, then, remember that I am *trying* to be fair and impartial. If you aren't convinced by the time I'm done that you should try our OSS products, I will know that I have managed to hide their true perfection well enough.

BY THE NUMBERS

In the chart accompanying this article (see *Figure 1*), I have listed fifteen computer languages and rated them in several categories (the best rating is a "1"). As I wrote this, some of the languages were not yet available for Atari ST computers, but at least one or two versions of each should make its appearance within the next year or so. I rated each language based on my estimates, and I must warn you that sometimes these are very subjective judgments. I have at least a minimal knowledge of the syntax and workings of each of these languages, but I certainly cannot claim to be a proficient programmer in each.

Some thoughts on the rankings themselves: My ratings are designed for beginner to intermediate programmers (six months to two or three years experience in a given language). In the hands of an expert any of these products might score higher, but expert programmers usually have their own favorite language, so this chart is not for them. Also, after I made this chart, I showed it to some other programmers. They convinced me to change a handful of rankings, but by and large they agreed that my rankings are all within one (1) of theirs. Plus or minus one, then, may be considered as my acceptable margin of error in this table.

Every language received a ranking for its relative speed, flexibility, and maintainability. Although these are based on my own personal subjective criteria, I would like to think that most professional programmers—except for the one-language fanatics—will accept them as reasonable (within my "acceptable margin of error").

- **Speed**—This does not refer to speed of compilation or program development; it is an estimate of the typical running time of a program written in that language. Languages with roughly equal run times get the same score. Generally, a language needs to be three or four times as fast as its competitors to get a better ranking. This is because variations in the efficiencies of the compilers (or interpreters) among manufacturers can often produce differences as great as this. Unlike the other categories, Speed refers only to language implementations for ST computers.

- **Flexibility**—Some languages clearly have one main purpose. COBOL is an excellent example; it really is not much good for anything other than business applications. Other languages are touted as general purpose. Obviously assembly language is the most flexible. Since all other languages eventually get reduced to its level, you can do anything with assembly that any other language can do. Watch out for the trap: Just because something is possible doesn't mean it is easy. I

couldn't even begin to want to write typical LISP programs in assembly.

- **Maintainability**—If you are the typical home hacker, this category may not be important to you. In a commercial environment, though, there is often nothing more important. After all, when your programming guru decides to chuck it all to grow grass in Mendocino, how do you fix his programs when you find bugs? (And I guarantee you will find bugs.) To me, even as an individual, maintainability is a consideration. I have often had the horrible experience of trying to figure out what a particular BASIC program does, three or four years after I wrote it.

A final note on maintainability: Any language can be written in a maintainable fashion. If nothing else, you can write a book to go along with the program, detailing line by line what each statement is doing. But some languages encourage self-documenting code and others discourage it. My rankings are based on my observations of code actually written and in use. If you do a better job, congratulations.

Sidelight: I strongly considered including an ease-of-use category, but I reluctantly left it out. The problem is that ease-of-use is often not a feature of a language but rather of a particular implementation (i.e., brand name) of that language. For example, I rate Atari 8-bit BASIC very easy to use, but ST BASIC gets only moderate marks. *Caveat emptor*; then, on this category. Take a good look at the actual product before buying, if possible. Read reviews, and don't be afraid to ask questions.

LUCKY SEVEN

After those three universal categories, I have rated each language for its appropriateness in each of seven categories. If a given language has no number in one of these columns, then

**No one
language is good for
all purposes.**

I think it is totally unsuitable for the purposes of this category.

Note that all these categories are somewhat an extension of the "flexibility" criterion. Presumably a flexible language should be usable for all seven tasks. But remember my caveat about assembly language: Just because you *can* do something in a particular language does not mean you would want to. Again, a brief description of each category follows:

- **Beginners**—This almost substitutes for the missing

Babel...

"ease-of-use" category, but to rank high here a language must be both easy to use and easy to understand. I surprised even myself by giving Prolog almost a top rating here, considering how much I struggled when I first encountered it. I now blame that on the lack of appropriate tutorial material. So let that serve as a warning. My ratings here assume that you are either taking a class from a competent instructor or you are learning from a good, available tutorial book.

- **Fun**—A most subjective category; it produced the greatest arguments from my fellow programmers. Sometimes I like to try out programming concepts just for the heck of it. How many of you remember my writing a BASIC interpreter in—of all things—BASIC? When I experiment I usually want an instant-response environment, so BASIC often wins. But for some more specialized concepts, I find it better to hack around in other languages. Hence this category and these rankings. The only way I'd consider an assembler or macro assembler fun is if their environment is good (e.g., integral editor and debugger). My personal choice for an ideal fun language would be a really good Pascal interpreter with a compatible optimizing compiler.

- **Business**—Is the language suitable for typical business applications, such as inventory control, general ledger, etc.? Can it be used for general-purpose database programming? I am constantly amazed at the number of business programs written in BASIC, even though it lacks most of the rudimentary tools I would want for such a purpose (indexed file access, user-definable variable types, highly structured I/O, just for starters).

- **Scientific**—Could I design a bridge with this language? Fortran has been the leader in this field for a long time, but there are signs that other languages may actually have more to offer. I think the reason Fortran is so accepted for scientific work is the incredible library of scientific functions that always accompanies it. But those functions are being written for other languages as well, especially those showing an asterisk in this column.

- **Action Games**—This is a catch-all category. Among other things, it implies the ability to do superior graphics. On the ST, a good rating here means that the language has structures to access the system's built-in graphics as well as high speed operations to move things around fast. Note that I have not allowed the use of assembly language subroutines in languages such as BASIC! If you constantly must resort to things outside a language, the language is deficient. Try another language. I do not preclude the use of libraries written in assembly language, but they should be included in the language package. The individual programmer should not have to write them. The asterisk on Forth in this column indicates that a compiled version of the language will rank at least one place higher.

- **Artificial Intelligence**—Once a very specialized category, it is becoming more and more popular. How easily can you tell your computer to "MOVE the blue pyramid on top of the red box"? The current crop of artificial intelligence languages are often rated as very slow, but this may be an illusion. How many lines of C code would it take you to make your computer perform that "MOVE" of a couple of sentences back? Maybe 10 or 20 times as many as a LISP or Prolog program. I would bet. Note that the "AI" languages are also good for some programs traditionally thought of as business applications, such as modeling and simulations. But this part of the AI field is in its infancy, so I haven't counted it too heavily.

- **Systems Software**—This is a pretty broad category. Operating systems, DOS utilities, computer languages, and more all fall here. Some languages perform adequately in all systems areas; others are limited to two or three tasks. Then, too, there is systems-software theory, where we investigate new ways of writing these products. Generally, the artificial-intelligence languages (denoted by an asterisk in this column) do well here. LISP, for example, has been used to write many more specialized languages, most of which you've never heard of. And the Japanese are using variations of Prolog to test operating system design theory in their multi-processor computers. (These languages fall down sharply, though, when we are talking about working systems—they are just too slow.) Anyway, I would bet that more practical, working systems software has been written in assembly, C, and Pascal than all other languages put together.

Don't expect me to try to justify each of the rankings I have given. Instead, I intend to concentrate on the best strengths (or worst weaknesses) of each language. Again, remember that I am not a professional in each of these languages, so expect more discussion of my favorites than of the others. But I promise to be opinionated even where I am ignorant.

BASIC

As a computer professional I am supposed to sneer and make nasty remarks about this language, but I must confess. BASIC has always been one of my favorites. Truthfully, it is not the language itself that is so appealing; rather, it is the BASIC environment that is fun to use. The whole idea of "instant programming" appeals to me. Imagine being able to modify a program almost as it runs! Got an error? Fix it and continue the operation. I also like to perform many programming quickies without writing a program at all! This is thanks to BASIC's ability to accept statements in "direct" mode, i.e., without line numbers as in conventional BASICs.

While I do not intend to comment on the quality of most available languages, I feel I must make a few statements about BASIC. Alas! The original ST BASIC from Atari is almost a joke. Imagine a language that can't even do five-digit arithmetic correctly. (Don't believe me? Try **PRINT 257*257**

sometime.) There are a few BASIC compilers available, but I view a BASIC compiler almost as an anachronism. If I want a compiled language, I will use a good one—not one as limited as BASIC! And BASIC is limited, despite what the diehards may tell you. Even though some BASICs have managed to add procedures with local variables, record-oriented I/O, etc., none of them has overcome the fundamental limitation of BASIC: lack of user-definable variable types.

Anyway, what I really want for the ST is a reasonably fast, interpreted BASIC with at least a reasonable complement of structured programming statements. It must both work within and program for the GEM environment. Atari has stated that they will be producing a new BASIC "real soon," and other companies are announcing interpreters even as I write this. So maybe, just maybe, I will be satisfied in the next few months, at the most. (Editor's note: See the sidebar to David Plotkin's article this issue for news of a new interpreted BASIC.)

And what about you? Well, the majority of computer owners never learn to program. And if they do dabble at it, they tend to try the languages which come free with the computer. If you fit in this category, I suggest you use Logo instead of the original ST BASIC. I may change this recommendation if and when Atari produces their new BASIC, but the current version is too buggy, too inaccurate, and too clumsy to bother with.

If you intend to spend some serious time learning to program, think hard before investing too much time in ST BASIC. If you already know some BASIC, then dabble away. But a serious application in ST BASIC? Anything business related? Forget it. Do remember, though, that these remarks are directed towards ST BASIC. Other BASICs may be more than adequate for many, many applications. ▶

FIGURE 1

Language Name	Speed Factor	Flexibility	Maintainability	Good for Beginners	Fun Time	Business Applic's	Science Applic's	Action Games	Artificial Intelligence	Systems
Interpreters										
BASIC	5	3	5	1	1	3	4	—	5	—
Logo	5	4	4	1	1	—	—	—	2	—
Prolog	6	3	3	2	2	—	—	—	1	1*
LISP	6	2	6	4	2	—	—	—	1	1*
DB2**	5	6	4	—	—	2	—	—	—	—
DB3**	4	2	2	5	3	1	—	—	—	—
Compilers										
Pascal	2	2	2	2	2	2	3*	3	3	3
Modula-2	2	2	2	3	3	2	3*	3	3	2
C	2	1	3	3	4	3	3*	2	4	1
Fortran	2	4	4	3	—	5	1	5	5	5
Ada	2	1	1	—	4	1	2	4	3	2
COBOL	4	6	1	2	—	1	—	—	—	—
Other										
Forth	3	2	7	3	3	—	—	4*	5	4
Assembler	1	1	4	—	6*	7	—	1	—	1
Macro-assembler	1	1	3	—	5*	6	—	1	—	1

*See text for explanation.

**DB2 and DB3 refer respectively to languages similar to dBase II and dBase III, products of Ashton Tate. Some such programs are available for the ST and more are expected.

Babel...

LOGO

This language was provided free with early units of the ST computers, so you may already be familiar with it. My 9-year-old can claim two years of experience with Atari Logo for the 8-bit machines. It's a fun language. You can make things move across the screen, draw pictures, and do most everything a 7-year-old would want to do. At 9, he's beginning to get bored with it.

Is that a commentary on Logo's capabilities? Not at all. A full-blown Logo can even be a reasonable artificial intelligence tool, and Logo for the ST is close to being full-blown (8-bit Logo is not). But to use these advanced capabilities requires advanced concepts, which are generally beyond the scope of Logo tutorials. Catch-22, right?

Still, learning the fundamentals of Logo may be one of the easiest and best introductions to computer programming possible. Don't expect to write many serious applications using Logo, but using the advanced functions will make the language useful for what I called "fun" programming.

PROLOG

When I first encountered Prolog, I thought it was bizarre and almost impossible to learn. Now that I have written a tutorial on the language, I have completely changed my mind. I have been able to teach Prolog to people who find BASIC cryptic! It is remarkably easy to learn the concepts of Prolog—perhaps even easier than learning Logo.

The neat part about Prolog is that it is also very powerful. True, finding the source of its real power takes experimentation and work, just as with Logo. But the end results are almost always worth it. Programs tend to be much simpler and much shorter than their equivalents written in a more "standard" language. See the discussions of the rankings for "Systems" and "Artificial Intelligence" for some pertinent comments.

Just a fair warning: If you already know an "algorithmic" language (e.g., C, Pascal, Fortran, BASIC, etc.), you may need to readjust your thinking before tackling Prolog. Generally, things that are easy in Prolog are difficult in those languages, and vice versa. Don't fall into the trap I did, be sure to learn from a good tutorial book. I am just egotistical and proud enough to say that the one by Mike Fitch and me is very good. But I will tell you of other good ones if you insist.

LISP

What can anyone say about LISP? It has been the darling of the artificial intelligence community for so long that it is required for computer science majors. Unfortunately, there is more than one *de facto* standard for the language itself. Xlisp, C-Lisp, InterLisp, and more all compete in this market. This isn't as bad as it sounds, though. Look how many dialects of

BASIC there are; the various LISPs are no more different from each other.

What about a LISP for a beginner? Only in a classroom situation, I think. Already have some experience with other languages? Then you might try tackling it with a good tutorial. LISP can even be a good "fun" environment, especially if you like to investigate languages and systems processes of several kinds. And a LISP in a real GEM environment could be quite powerful.

DB2 AND DB3

As the footnote to the rankings table shows, these are my appellations for various languages which are more or less compatible with Ashton-Tate's dBase II and dBase III. Until something better comes along for the ST, if you want to write database applications, these are the ones to use. DB3 is the successor to DB2, and it answers many of my objections to the latter, such as lack of subroutines, limited number of variables, only two files accessible simultaneously, and more.

**You can
do anything with assembly
language that any other
language can do.**

In a class environment with the right instructor, DB3 might even be a viable first computer language, though I think one really needs to have learned a more conventional language first. The DB2 environment was designed for machines which are tiny by ST standards, so it suffers in comparison. I would choose DB2 only if I had several dBase II programs that I wanted to move over to the ST.

PASCAL

Perhaps the greatest advantage of Pascal is that it is so standardized. Consider the fact that the programmers at Omnitrend managed to take over 1.5 million bytes of Pascal source code from an IBM PC, transport it to a certain Pascal on the ST, and get the game Universe II translated in a few weeks. To me, that is incredible portability.

There is a vast body of literature written about Pascal. Almost all schools, from junior college to the most prestigious universities, teach Pascal as a standard language. It is the only language that high school students may use to qualify for advanced college placement via the college board exams. The

list goes on. (Besides, almost a half million Turbo Pascal owners can't be completely wrong, can they? And Turbo Pascal isn't even particularly powerful as Pascals go.)

Does this mean that Pascal is perfect? Of course not. No language is. Many C programmers sneer at Pascal as being too limiting—they claim that it restricts their freedom of programming. Pascal, like BASIC, does such nice things for you as checking to be sure you didn't try to store something into the 233rd element of a 40-element array. (In C, and indeed in many loose languages, no such checks are performed. If you want to wipe out memory, go do it. Most modern Pascals will allow you to turn off the checks if you really must, so you can wipe yourself out with Pascal, as well.)

I maintain that Pascal can be an ideal beginner's language. It may even be better to learn Pascal before BASIC, especially if you have access to a local class or a good tutorial book. Does this mean that Pascal is not for professionals? Ask the ones who wrote Universe II about that. Overall, Pascal is a nice, safe choice for many, many people. Including me, of course.

MODULA-2

Almost everything I said about Pascal applies equally well to Modula-2. Since both owe their existence in part to a single person (Niklaus Wirth), this is not surprising. Wirth designed Modula-2 to alleviate some of the problems associated with the original Pascal. In particular, Modula-2 is designed to be written in "modules" and then linked together. Since the original Pascal compilers insisted on compiling an entire program at once, this should give Modula-2 an enormous advantage, especially on memory-limited microcomputers.

Unfortunately for Modula-2, by the time it appeared, most Pascal compilers already supported many of its features, including (you guessed it) modular compilation. The result is that my rankings show it virtually dead even with Pascal. It loses a bit for beginners for just one reason: It is newer, so fewer tutorials are available and fewer schools teach it. Feel just a little bit adventuresome? This might be a reasonable choice.

C

If Pascal and Modula-2 are too limiting, then C is an anarchist's dream. There are no holds barred when using C. Want to twiddle the bits in a hardware register? C makes it easy. Want to chuck the system I/O library and write your own? Since C has no built-in I/O, it is a natural choice for this!

I used to be a C hacker. I still like the language, but I think that I have grown out of the hacker stage and am ready to admit that the most important aspect of a program is not how fast it runs. Thanks to a unique macro preprocessor, C source code can be as maintainable as any other language. And when

I write some C code, I like to think that any competent programmer could maintain it. Unfortunately, this same flexibility can lead to totally unreadable code. If you use C, it is your choice.

Some C fanatics claim that C compilers inherently generate better code than other languages (usually they are pointing the finger at Pascal or Modula-2). This is nonsense. What is really happening is that almost none of the compilers for microcomputers do very good code optimization. There is economic justification for this: Would you pay three to ten times the price for a compiler which generated 50 percent better code? Probably not. Yet a good optimizing compiler is much, much more expensive to write. A really good optimizing compiler replaces the need for writing torturous C code. Even with this help, C compilers aren't necessarily much (if any) faster than other languages, as my table reflects. (As a specific example, the highly touted Megamax C runs the "Dhrystone" benchmark only a couple of percent faster than does Personal Pascal.)

Bickering aside, C is a good, general purpose language. Since it was used to write major portions of the ST's operating system, it already has an "in" of sorts. But for beginners? I don't think so. The lack of bounds checking (see the Pascal discussion) is so often disastrous, that it can be pure frustration. On the IBM PC, several interactive C interpreters (with bounds checking) have become available. When one appears for the ST, consider learning C with it.

ADA

What can I say about a language that isn't even available? Well, if you want to win government contracts, learn Pascal or Modula-2 or C and wait for the first Ada compilers. But don't hold your breath. Ada is a fantastically complicated language, and it takes a while to certify a compiler to government standards. When the first full compilers appear, expect them to cost about as much as your computer. Also expect a 20-megabyte hard disk to be a minimum storage requirement.

Ada is undeniably powerful. You can not only define your own variable types, you can even define your own operators to work on those types. Want to find the number of days between two dates where you have specified the record format of a date? Just define the proper kind of "-" (minus) operator. No way is this a beginner's language!

COBOL

COBOL is an acronym for COmmon Business Oriented Language. And if you want to write industry-standard business applications, if you want portability between your ST and your mainframe at work, and if you love to write self-documenting code, buy COBOL! What other language lets you write code that is as readable as this: "Add Price of Object to Sales-tax Giving Subtotal."

Babel...

And, since COBOL is taught in virtually every vocational school and junior college in the country, it's even a pretty good choice for beginners. (Look how many companies advertise on TV that they can turn any high school graduate into a programmer. Guess what language they teach?) More commercially successful programs have been, and still are, written in COBOL than in any other language; so, if you are looking for a job in programming, you have to at least think about this language. Just be forewarned that thousands of others have got the same idea.

FORTH

Forth is an enigma of sorts. The language itself is a little strange to those of us who think algebraically, but it is fairly consistent and not too difficult to learn. The reason that I am hesitant to recommend Forth any more strongly is because of its environment. It was originally designed to be both language and operating system on 8,000 byte computers, and the Fig Forth standard shows this. ("Fig" should be "FIG" since it is an acronym for "Forth Interest Group," but the Fig's want it this way.)

Programs edited by "screens," with each screen limited to 1024 bytes? Come on now! I've got 520,000 bytes to play with here. Why should I be limited to this kind of obsolete environment? Worse, disk I/O is supposed to take place via screens, also. Forget the operating system, just give me 20,000 numbered screens on my hard disk. Ludicrous! I think the Forth Interest Group has done more to hinder the development of this language than anyone has done to enhance it.

Now that I have all the Forth fanatics up in arms, let me retreat just a little. Luckily, the better Forths don't stick to the Fig standard. Most of them use the Fig version as a kind of internal starting place and then build better structures on top of that. Such things as file I/O, GEM-based program editing, and much more contribute to make Forth a more viable language.

As to capabilities, Forth is no slouch. Thanks to its unique "threaded" design, it interprets Forth as fast as compiled programs in some other languages run (COBOL, as a specific example). And Forth compilers are sometimes available to improve performance even more.

Unfortunately, I don't think that Forth can be said to be the best at anything of importance. It used to win on compactness of code, but advanced processors and large memory sizes have dulled that issue. It gets fair marks for beginners because of its semi-interactive nature. Some versions of it might even get a better mark here. Perhaps the most telling point is the paucity of commercial products written in Forth. If Forth is as good as the fanatics claim, why don't more developers use it? (Yes, yes, I know of some significant exceptions—H&D's dBase II clone, for example—but the statement still stands.) I think I could like the right version of Forth. Let me know if you find one.

ASSEMBLER AND MACRO ASSEMBLER

When it comes right down to it, there are some things that you just can't do as well in any higher-level language as you can do in assembler. Let's face it, no matter how good a compiler is, it can't possibly produce code which runs faster than assembler (equal to, perhaps, but not faster—by definition). But on the ST, about the only applications which I think can justify the extra work of assembly language are high-speed graphics programs. In other words, games. (Okay, okay. . . maybe 3-D drawing programs and the like. . . but how many of you are going to write one of those? How many want to write a game? I rest my case.)

Now I happen to really like assembler. If I had to be confined to two languages, I might choose a really good BASIC (for all my quick and dirty stuff) and a good macro assembler (for everything else). It's gotta be a macro assembler, though. As complex as the interface to GEM is, I would get tired awfully quickly if I had to code each call in "longhand." A powerful macro assembler can make writing some programs almost as easy as using a high-level language. (What the heck. . . one more dig: I'll bet I could write a set of macros to duplicate 90 percent of the common Forth functions.) Besides, a program written using lots of macros can be very self-documenting.

Sigh. Unfortunately, so far, I haven't seen the macro assembler for the ST. If you are familiar with MAC/65 for the 8-bit Ataris, you will begin to see what I want. I need more powerful macros than MAC/65 gives (because of the complex GEM interface) and a GEM-based editor, and. . . but the idea is right. Editor, macro assembler, debugger which doesn't affect my program's screen or keyboard I/O, DOS commands—it's just not there yet. And my "fun" rankings reflect that. I would move a system such as I have described up at least one place in this category.

A warning: Beginners, stay away from assembly language! On the 8-bit machines, you could possibly start with assembler if you had a good tutorial (of which there were none for Atari, unfortunately). On the ST? I don't think so, unless you are willing to limit yourself to piddly little programs. You have to understand GEM as it is practiced in one of the higher level languages before you tackle it at the assembly language level.

WRAPPING UP

Am I really done? Hard to believe! If you are thoroughly confused, maybe I have succeeded. If nothing else, I would like you to read reviews, check with your local users group, and get better educated generally. Try to stay away from the fanatics. If a person doesn't program regularly in at least two languages, be suspicious of any advice. No one language is good for all purposes. ■

From a book:
"First Book of Atari Graphics"

5 Animation With Player Missile Graphics

Introduction To Player/Missile Graphics

Bill Wilkinson

This article describes the features of the Atari player/missile graphics system ("P/M graphics" for those of us with lazy typing fingers). Although there are now other systems available with similar capabilities (notably "sprites" on the Commodore and Texas Instruments TI 99-4A computers), there are several aspects of P/M graphics which are uniquely and powerfully Atari.

For reasons having to do with lack of information and (often) an abundance of misinformation, many Atari owners think of players and missiles as some mysterious aspect of the machine which requires convoluted machine language and arcane rites to control properly. In truth, P/M graphics is in many ways less mysterious than the standard Atari "playfield" graphics. Have you yet truly deciphered the relationship between SETCOLOR and COLOR? (I haven't. I usually use trial and error to find the connection that I need.) Have you mastered the concept of display lists? (I didn't ask if you could produce one, just if you understood the level of indirection that is needed to produce even the simplest display on an Atari.) Player/missile graphics is actually *simple* compared to some of these obscurities.

I think the first thing needed to understand P/M graphics is a little flexibility in conceiving how memory is mapped into display in the Atari computer. Consider Figure 1. As far as the Central Processing Unit (CPU) or most of the Atari hardware is concerned, memory is simply one long string of bytes. (Well, some parts of the system like to digest memory in 1K, 2K, or 4K byte blocks, but within those blocks it's all a string of bytes.) But if you look at Figure 1, you will probably soon decide that this is not a reasonable way for human beings to consider mem-

5 Animation With Player Missile Graphics

ory, especially human beings who are trying to visualize memory being displayed as graphics.

So consider instead what we know of BASIC graphics mode 19 (GRAPHICS 3 + 16). GRAPHICS 19 (which is simply a full-screen version of GRAPHICS 3) consists of 24 lines of 40 pixels each, where each pixel occupies only two bits of memory, implying that a line is only 10 bytes long. Instead of thinking of memory as one long string of bytes, why not consider it as an array of ten-byte strings? This visualization is presented in Figure 2, and you will probably agree that this is a much clearer representation than that of Figure 1.

One more exercise before leaving this subject: try visualizing the normal text (GRAPHICS 0) display screen as a representation of memory. How many lines are there? How many bytes per line? I hope you answered 24 lines of 40 bytes each; a pictorial representation of this display mode is shown in Figure 3.

So just exactly what is a player or a missile? First, let's note that for most purposes there's no real difference between a player and a missile other than size, so all further references to "players" may be assumed to refer to missiles also, unless otherwise noted. A player, then, is simply the graphic video display of a portion of the Atari computer's main memory. "So what?" you say. "That's how all computers put stuff on the screen: by displaying the memory." True. And we just showed diagrams of how the Atari also displays its main video screen from what Atari calls "playfield memory." But players and missiles are displayed independently of the playfield and from an entirely separate segment of memory.

The "S:" ("Screen") device driver, which is what actually processes such BASIC keywords as GRAPHICS, PLOT, and DRAWTO, knows nothing of P/M graphics. In a sense this is proper: the hardware mechanisms that implement P/M graphics are, for virtually all purposes, completely separate and distinct from the "playfield" graphics supported by "S:". For example, the size, position, and color of players on the video screen are completely independent of the GRAPHICS mode currently selected and any COLOR or SETCOLOR commands currently active. In Atari parlance, a "player" is simply a contiguous group of memory cells displayed as a vertical stripe on the screen.

We again take refuge in a diagram, that of Figure 4. This figure shows a standard playfield display along with that por-

tion of Ram
the display
character s
piece of me
of the scre
that the pl
character s
is *always* tr
wide array
graphics m
is which?),
to thinking

The "a
sure we are
Second, th
instead of
accurate on
Player 1 sta
Third, ther
the amount
"semi-fixed
(equivalen
bytes each,
valent to C
each. But e
simple, sin

Sound
five, dependen
in any of th
for specific
player's di
correspon
all! That is,
derlying p

Why c
idea of the
bounds of
ones as sho
doing a cir
senting the
something
again? First

5 Animation With Player/Missile Graphics

tion of Random Access Memory (RAM) being used to generate the display (and note the representation of RAM as a series of character strings). But notice that the figure also shows another piece of memory being used to display something else on part of the screen. This "something else" is a player. And notice that the player's portion of RAM is shown as an "array" of character strings where each string is only one byte long. This is *always* true: *all* players are always displayed as a one-byte wide array. There are no display lists to worry about, no graphics modes (using 10 or 20 or 40 bytes per line, and which is which?), and no visualization problems. It's like being back to thinking of memory as one long string of bytes – almost.

The "almost" is the kicker. First, note that we need to make sure we are thinking of the string as being stacked vertically. Second, the pictorial representation (the player on the screen instead of the string of bytes in memory) is actually the more accurate one, since each player is always a "semi-fixed" length: Player 1 starts on the very next byte after Player 0, and so on. Third, there are actually two choices open to the user regarding the amount of memory used by each player (hence the words "semi-fixed"): players may have very fine vertical resolution (equivalent to GRAPHICS 8), in which case they occupy 256 bytes each; or they may have relatively coarse resolution (equivalent to GRAPHICS 7), in which case they occupy 128 bytes each. But even with this minor complication, players are fairly simple, since all players must always have the same resolution.

Sounds dull? Consider: each player (and there are four or five, depending on how you think of missiles) may be "painted" in any of the 128 colors available on the Atari (see SETCOLOR for specific colors). Within the vertical stripe which is each player's display, each bit set to 1 paints the player's color in the corresponding pixel, while each bit set to 0 paints no color at all! That is, any 0 bit in a player stripe has no effect on the underlying playfield display.

Why call it a vertical stripe? Refer to Figure 5 for a rough idea of the player concept. If we define a shape within the bounds of this stripe (by changing some of the player's bits to ones as shown), we may move the player vertically by simply doing a circular shift on the contiguous memory block representing the player. Why is that easier than simply PLOTting something on the playfield and then moving it by PLOTting it again? First, since the player does not affect the playfield, any

5 Animation With Player Missile Graphics

pretty picture (or text or whatever) on the main screen remains unchanged. Second, because it's a lot easier to do a circular shift on a byte string than it is to change memory cells that are 40 (or 20 or 10?) bytes apart in memory.

Finally, the real clincher: even though vertical movement requires some shuffling of a string of bytes, horizontal movement is essentially effortless. Each and every player *and* missile has its own independent *register* (i.e., memory location) which controls its current horizontal position on the screen. Moving a player stripe horizontally is as easy as a single POKE from BASIC.

To summarize and simplify: A player is actually seen as a stripe on the screen eight pixels wide by 128 (or 256) pixels high. Within this stripe, the user may POKE or move bytes to establish what is essentially a tall, skinny picture (though much of the picture may consist of 0 bits, in which case the background "shows through"). Using a simple POKE, the programmer may then move this player to any horizontal location on the screen. To move a player vertically, though, one must do some sort of shift or move on the contents of the string of bytes displayed in the stripe.

From standard Atari BASIC, there is no easy way to move these stripes vertically (using a FOR/NEXT loop with PEEKs and POKEs is simply too slow). And while there now exist languages which have built-in mechanisms to do this movement (e.g., MOVE in Microsoft BASIC; MOVE and PMMOVE in BASIC A+), the overwhelming use of Atari BASIC has prompted many authors to try their hands at providing this movement in a form easily usable from Atari BASIC. This chapter includes sections detailing a few of the methods which have been worked out.

And now some final comments before we leave this introduction to player missile graphics:

Missiles pretty much work just like players except that (1) they are only two bits wide instead of eight, (2) all four missiles share the same 128 or 256 bytes of memory (each using only its own bits in each byte), (3) each two-bit sub-stripe has an independent horizontal position register, and (4) by default a missile has the same color as its parent player. A later section in this chapter will delve a bit deeper into the mysteries of missiles.

There are essentially only five primary controls available to the P/M graphics user. We have already mentioned three: inde-

pendent
depende
trol over
simply "

In ac
dently co
specified
width. T
used for
pixel. Re
showing

Incide
vertical re
size as th
dental ha

The l
where in
The rule i
solution (i
byte mem
per playe
boundary

Are y
can you p
1K bytes l
five, you
the Atari
allocate th
see why.

Do y
No. There
or what-h
the langua

And
herein; th
become o
more thin
you about

5 Animation With Player/Missile Graphics

pendent control of the various player horizontal positions, independent control of the player's colors, and system-wide control over the resolution (128 bytes or 256 bytes per player, or simply "off").

In addition, each player and each missile has an independently controllable "width." A player or missile may be specified as single-width (narrow), double-width, or quadruple-width. This width does *not* affect the number of bytes or bits used for the display; it affects only the width of each individual pixel. Refer to Figure 6 for a diagram of a four-player system showing independent horizontal position and width control.

Incidentally, single-width players generated in the 128-byte vertical resolution mode have square pixels which are the same size as those in GRAPHICS 7, a presumably not altogether accidental happening.

The last control available to the user is the ability to specify where in memory the player and missile stripes are to be located. The rule is fairly simple: you need 2K bytes for single-line resolution (256 bytes per player), and it must be located on a 2K-byte memory boundary. For double-line resolution (128 bytes per player), you need a 1K-byte segment located on a 1K byte boundary.

Are you quick in arithmetic? How many 256-byte players can you put into 2K bytes? Or how many 128-byte players can 1K bytes hold? If you answered eight, you pass. If you answered five, you can go to the head of your Atari class. Indeed, with the Atari P/M memory map, you "waste" three players if you allocate the full amount of memory called for; see Figure 7 to see why.

Do you see the wasted memory? Does it need to be wasted? No. There is no reason why you can't put data, character sets, or what-have-you in this area. Indeed, in BASIC A+, part of the language is in this otherwise excess area.

And now you are ready to peruse the secrets unveiled herein; the darkest mysteries of player/missile graphics will become open to you. But don't be surprised if you find even more things that can be done with P/M graphics than we told you about here.

5 Animation With Player/Missile Graphics

Figure 1. RAM considered as a linear “string” of bytes.

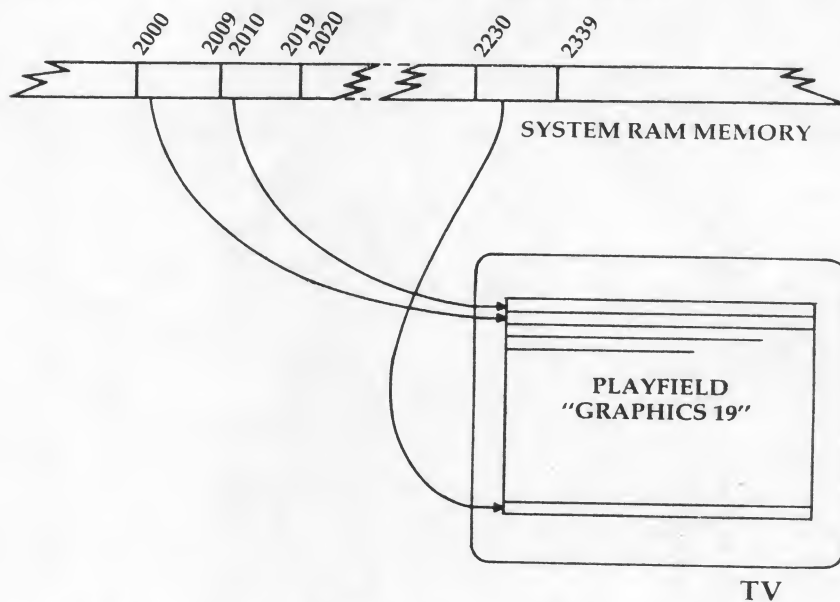
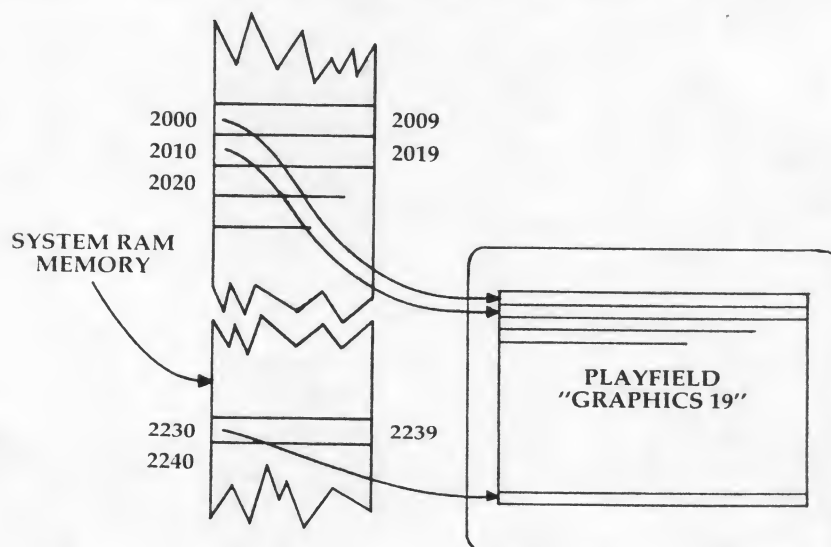


Figure 2. RAM considered as an array of 10-byte strings.



5 Animation With Player/Missile Graphics

Figure 3. RAM considered as an array of 40-byte strings.

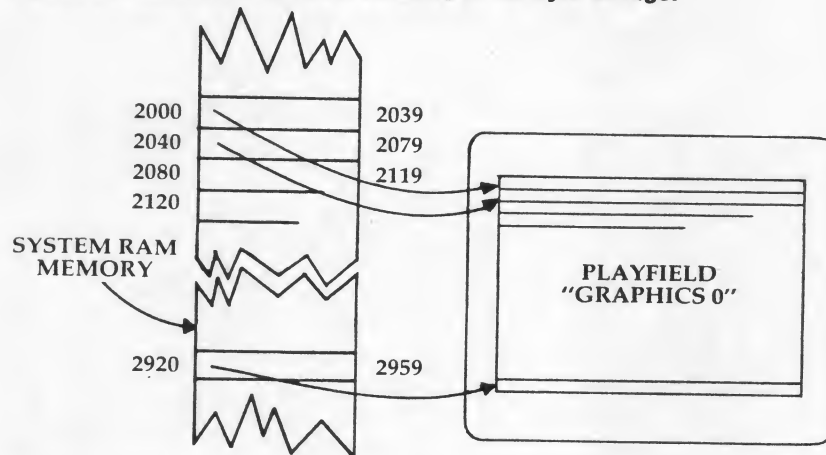
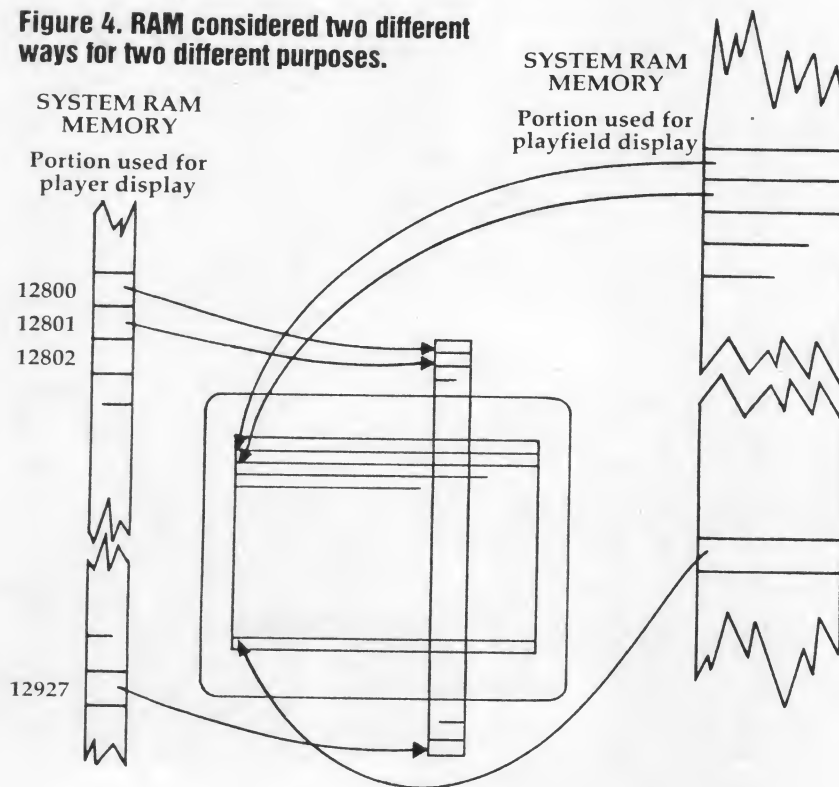
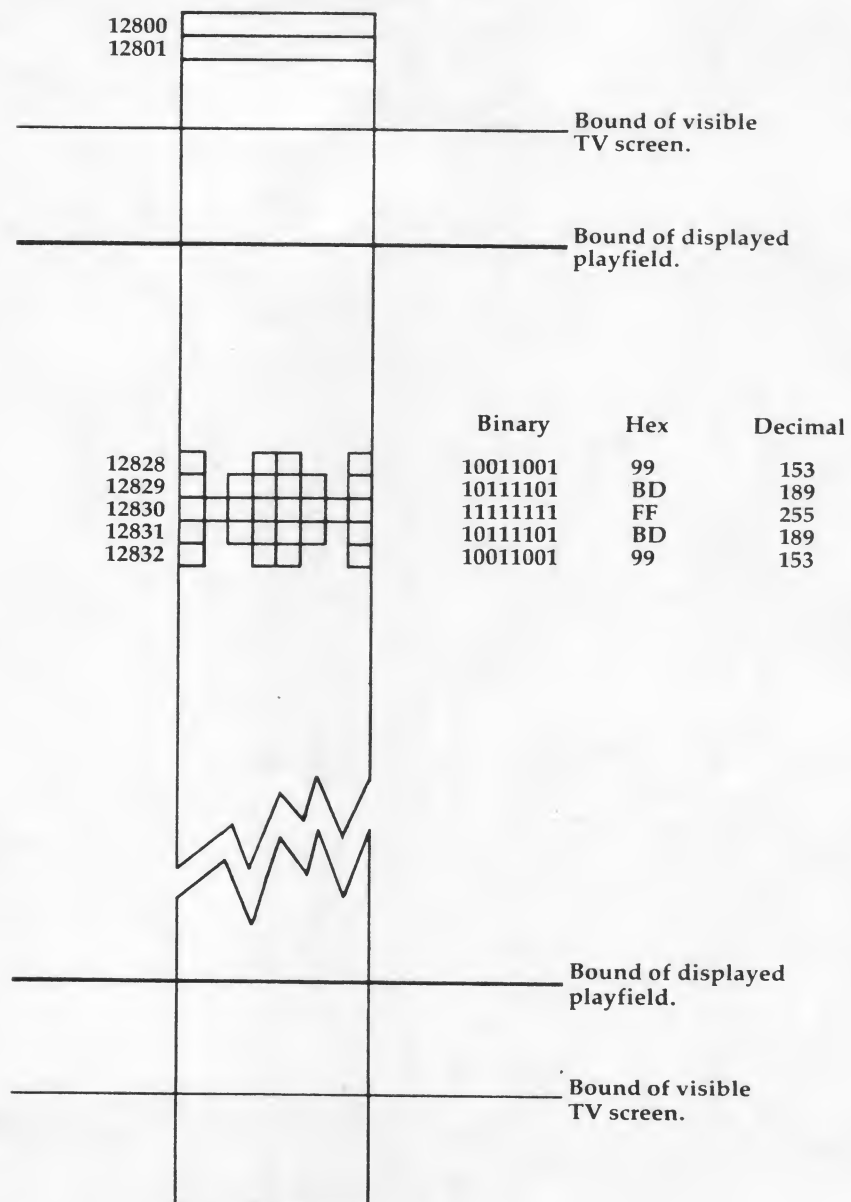


Figure 4. RAM considered two different ways for two different purposes.



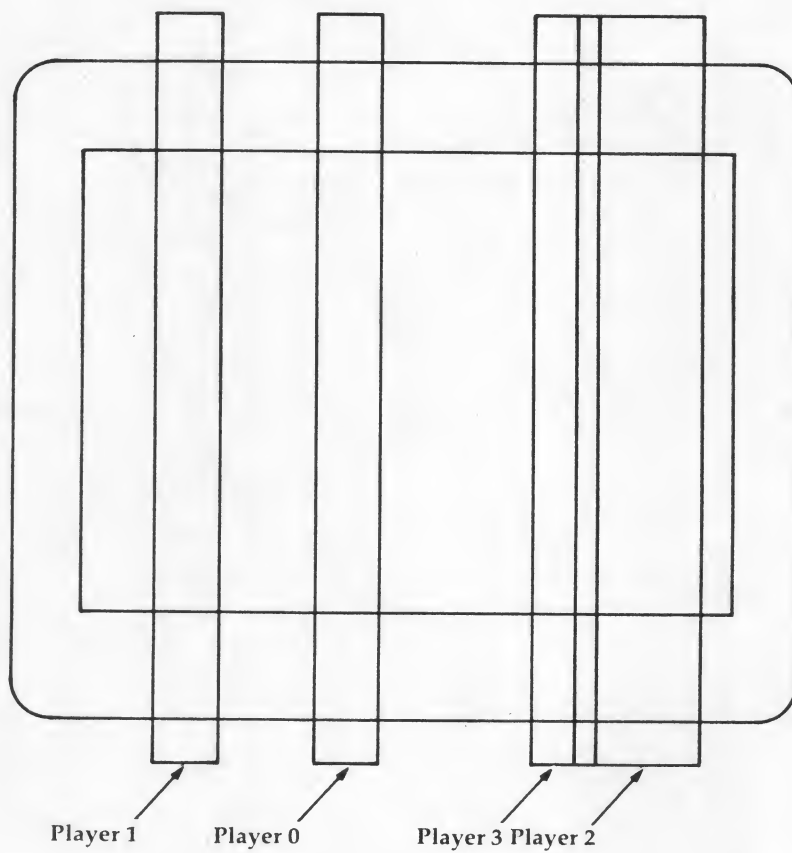
5 Animation With Player/Missile Graphics

Figure 5. Detail – the display of Player Memory.



5 Animation With Player/Missile Graphics

**Figure 6. Four Players at once.
(Player 2 is double width and overlaps Player 3)**



Atari BASIC:

A High-Level Language Translator

The programming language which has become the *de facto* standard for the Atari Home Computer is the Atari 8K BASIC Cartridge, known simply as Atari BASIC. It was designed to serve the programming needs of both the computer novice and the experienced programmer who is interested in developing sophisticated applications programs. In order to meet such a wide range of programming needs, Atari BASIC was designed with some unique features.

In this chapter we will introduce the concepts of high level language translators and examine the design features of Atari BASIC that allow it to satisfy such a wide variety of needs.

Language Translators

Atari BASIC is what is known as a *high level language translator*.

A *language*, as we ordinarily think of it, is a system for communication. Most languages are constructed around a set of symbols and a set of rules for combining those symbols.

The English language is a good example. The symbols are the words you see on this page. The rules that dictate how to combine these words are the patterns of English grammar.

Without these patterns, communication would be very difficult, if not impossible: Out sentence this believe, of make don't this trying if sense you to! If we don't use the proper symbols, the results are also disastrous: @twu2 yeggopt gjsiem, keorw?

In order to use a computer, we must somehow communicate with it. The only language that our machine really understands is that strange but logical sequence of ones and zeros known as machine language. In the case of the Atari, this is known as 6502 machine language.

When the 6502 central processing unit (CPU) "sees" the sequence 01001000 in just the right place according to its rules of syntax, it knows that it should push the current contents of

from a book!
Basic Source Book
"Atari"

the accumulator onto the CPU stack. (If you don't know what an "accumulator" or a "CPU stack" is, don't worry about it. For the discussion which follows, it is sufficient that you be aware of their existence.)

Language translators are created to make it simpler for humans to communicate with computers. There are very few 6502 programmers, even among the most expert of them, who would recognize 01001000 as the push-the-accumulator instruction. There are more 6502 programmers, but still not very many, who would recognize the hexadecimal form of 01001000, \$48, as the push-the-accumulator instruction.

However, most, if not all, 6502 programmers will recognize the symbol PHA as the instruction which will cause the 6502 to push the accumulator.

PHA, \$48, and even 01001000, to some extent, are translations from the machine's language into a language that humans can understand more easily. We would like to be able to communicate to the computer in symbols like PHA; but if the machine is to understand us, we need a language translator to translate these symbols into machine language.

The Debug Mode of Atari's Editor/Assembler cartridge, for example, can be used to translate the symbols \$48 and PHA to the ones and zeros that the machine understands. The debugger can also translate the machine's ones and zeros to \$48 and PHA. The assembler part of the Editor/Assembler cartridge can be used to translate entire groups of symbols like PHA to machine code.

Assemblers

An assembler — for example, the one contained in the Assembler/Editor cartridge — is a program which is used to translate symbols that a human can easily understand into the ones and zeros that the machine can understand. In order for the assembler to know what we want it to do, we must communicate with it by using a set of symbols arranged according to a set of rules. The assembler is a translator, and the language it understands is 6502 assembly language.

The purpose of 6502 assembly language is to aid program authors in writing machine language code. The designers of the 6502 assembly language created a set of symbols and rules that matches 6502 machine language as closely as possible.

This means that the assembler retains some of the

disadvantages of machine language. For instance, the process of adding two large numbers takes dozens of instructions in 6502 machine language. If human programmers had to code those dozens of instructions in the ones and zeros of machine language, there would be very few human programmers.

But the process of adding two large numbers in 6502 assembly language also takes dozens of instructions. The assembly language instructions are easier for a programmer to read and remember, but they still have a one-to-one correspondence with the dozens of machine language instructions. The programming is easier, but the process remains the same.

High Level Languages

High level languages, like Atari BASIC, Atari PILOT, and Atari Pascal, are simpler for people to use because they more closely approximate human speech and thought patterns. However, the computer still understands only machine language. So the high level languages, while seeming simple to their users, are really much more complex in their internal operations than assembly language.

Each high level language is designed to meet the specific need of some group of people. Atari Pascal is designed to implement the concept of structured programming. Atari PILOT is designed as a teaching tool. Atari BASIC is designed to serve both the needs of the novice who is just learning to program a computer and the needs of the expert programmer who is writing a sophisticated application program, but wants the program to be accessible to a large number of users.

Each of these languages uses a different set of symbols and symbol-combining rules. But all these language translators were themselves written in assembly language.

Language Translation Methods

There are two different methods of performing language translation — *compilation* and *interpretation*. Languages which translate via interpretation are called *interpreters*. Languages which translate via compilation are called *compilers*.

Interpreters examine the program source text and simulate the operations desired. Compilers translate the program source text into machine language for direct machine execution.

The compilation method tends to produce faster, more efficient programs than does the interpretation method. However, the interpretation method can make programming easier.

Problems with the Compiler Method

The compiler user first creates a program source file on a disk, using a text editing program. Then the compiler carefully examines the source program text and generates the machine language as required. Finally, the machine language code is loaded and executed. While this three-step process sounds fairly simple, it has several serious "gotchas."

Language translators are very particular about their symbols and symbol-combining rules. If a symbol is misspelled, if the wrong symbol is used, or if the symbol is not in exactly the right place, the language translator will reject it. Since a compiler examines the entire program in one gulp, one misplaced symbol can prevent the compiler from understanding any of the rest of the program — even though the rest of the program does not violate any rules! The result is that the user often has to make several trips between the text editor and the compiler before the compiler successfully generates a machine language program.

But this does not guarantee that the program will work. If the programmer is very good or very lucky, the program will execute perfectly the very first time. Usually, however, the user must debug the program.

This nearly always involves changing the source program, usually many times. Each change in the source program sends the user back to step one: after the text editor changes the program, the compiler still has to agree that the changes are valid, and then the machine code version must be tested again. This process can be repeated dozens of times if the program is very complex.

Faster Programming or Faster Programs?

The interpretation method of language translation avoids many of these problems. Instead of translating the source code into machine language during a separate compiling step, the interpreter does all the translation *while the program is running*. This means that whenever you want to test the program you're writing, you merely have to tell the interpreter to run it. If things don't work right, stop the program, make a few changes, and run the program again at once.

You must pay a few penalties for the convenience of using the interpreter's interactive process, but you can generally develop a complex program much more quickly than the compiler user can.

However, an interpreter is similar to a compiler in that the source code fed to the interpreter must conform to the rules of the language. The difference between a compiler and an interpreter is that a compiler has to verify the symbols and symbol-combining rules only once — when the program is compiled. No evaluation goes on when the program is running. The interpreter, however, must verify the symbols and symbol-combining rules every time it attempts to run the program. If two identical programs are written, one for a compiler and one for an interpreter, the compiled program will generally execute at least ten to twenty times faster than the interpreted program.

Pre-compiling Interpreter

Atari BASIC has been incorrectly called an interpreter. It does have many of the advantages and features of an interpretive language translator, but it also has some of the useful features of a compiler. A more accurate term for Atari's BASIC Language Translator is *pre-compiling interpreter*.

Atari BASIC, like an interpreter, has a text editor built into it. When the user enters a source line, though, the line is not stored in text form, but is translated into an intermediate code, a set of symbols called *tokens*. The program is stored by the editor in token form as each program line is entered. Syntax and symbol errors are weeded out at that time.

Then, when you run the program, these tokens are examined and their functions simulated; but because much of the evaluation has already been done, the execution of an Atari BASIC program is faster than that of a pure interpreter. Yet Atari BASIC's program-building process is much simpler than that of a compiler.

Atari BASIC has advantages over compilers and interpreters alike. With Atari BASIC, every time you enter a line it is verified for language correctness. You don't have to wait until compilation; you don't even have to wait until a test run. When you type RUN you already know there are no syntax errors in your program.



Avoiding Disk Errors

I know many of you will find this hard to believe, but I've never encountered a disk error on the Atari which I couldn't explain. Further, I have had very few DOS errors, ever. (The reasons for the few errors I have encountered, by the way, were always related to random access files—a common problem with Atari DOS 2.0 and its derivatives.) Yet after a few hundred phone calls and letters, I know that many of you have experienced the frustration of wiped-out disk files. Why? Well, I can't know each and every reason, and I can't repair damage that's already been done, but maybe I can give you some helpful hints for the future.

Hands Off That Disk

Hint 1: Never, never, never take a disk out of a drive unless the program you're using tells you to. (This goes beyond even the good advice about never removing a disk when the drive is still spinning.) In particular, never swap disks until prompted to do so. Why? Well, because the Atari disk drive has no way to tell the computer that the disk has been removed or changed.

Consider: How does any DOS know what disk sectors to allocate to a new file? Generally, a DOS keeps a list of unused disk sectors. The next time it needs to find a sector (for example, to extend a file), it takes one from this list. The list (called a *Volume Table Of Contents* or VTOC in Atari parlance) is usually kept on disk until a file is opened, when it is read into memory. It is rewritten to the disk when a file is closed.

Okay, now open a file for output, write some information, swap disks, and write more data. What happens? The list of sectors was correct for the first disk, but it's extremely unlikely that it bears any reasonable relation to what exists on the second disk. Most probably,

DOS will allocate several sectors which were already part of other files on the second disk. Kablooeey!

If you're using an application program, then, follow the prompts and don't swap disks unless told to. If you're programming and working with disk files, make sure you close all open files before swapping disks (END automatically closes all open files in BASIC). If you're using DOS, you should be safe as long as you change disks only at the DOS prompt. Of course, when duplicating files or disks, you must swap disks when DOS tells you to.

Beware Of RESET

Hint 2: Never hit the SYSTEM RESET button during a disk operation. For example, if you hit RESET in the middle of a SAVE, it's possible to end up with a completely blown program. In fact, if you then SAVE the program to disk again, you could end up with a blown disk file.

This results from a really subtle bug in DOS 2.0. When DOS enters what is known as *burst I/O* mode (to speed up input/output), it "copies" the memory to disk. But DOS 2.0's file organization requires that the last three bytes of each sector contain a link to the next sector in the file. How can it do this when it is writing directly from memory? Answer: By "swapping" three bytes of memory long enough to write a sector, and then restoring the bytes.

Now suppose you happen to hit RESET when those three bytes are swapped out. Oops...say goodbye to your program.

There are two ways to fix this problem. First, since DOS gets control after a RESET, it could check to see if a disk write was interrupted. If so, it could restore the three bytes. Or, second, DOS could always copy bytes to be written into a buffer, thus never disturbing the program (or data) in memory. The

second approach is successfully used by DOS 2.5.

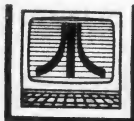
Missing Sectors

Hint 3: Avoid hitting RESET during disk operations even if you're using DOS 2.5, because you may still mess up the disk a bit. Here's one way: Open a file for write (OPEN #5,8,0,"D:FILE" in BASIC, for example), write some data, OPEN another file for write, write data to both files, CLOSE the first file, write some more data to the second file, and then hit RESET. What happens?

The VTOC says the sectors in the second file which were written before the CLOSE are now in use (and that was true when the CLOSE took place). If you add the number of free sectors remaining on the disk to the number of sectors used in all files, the total is no longer 707 in single density or 1010 in enhanced density, as it should be. You just lost some of your disk space.

Hint 4: Everything I just mentioned about RESET also applies to turning off the power. For example, if you have a power failure in the middle of a SAVE from BASIC or while there are some data files open in a business program, be prepared for some problems.

Fortunately, DOS 2.5 comes with a program called DISKFIX.COM which does a pretty good job of fixing up a "damaged" disk (either DOS 2.0 or 2.5). It allows you to undelete files as long as you haven't written any new files since the deletion. At your choice it will either try to recover or permanently remove a file which was left open for output. And, most importantly, it checks each file on the disk to make sure it is OK, and then reconstructs the VTOC to ensure that all 707 or 1010 sectors are accounted for. ☐



Do You Need A 16-Bit Computer?

There has been a disturbing trend in my reader mail for the last couple of months. On the one hand, more and more people are asking for help: Where can I find out how to work with player/missile graphics? How do I hook a model 2300 argon laser to an Atari's joystick ports and shoot down unfriendly flying saucers? (That's not as much an exaggeration of the original question as you may think.) At the same time, and all too often from the same people, I hear of grandiose plans to buy an Atari ST or an Amiga and make the world safe for computocracy. I hate to burst any bubbles, but let's reason together for a moment.

Over the past six years there have been at least 60 or 70 books published about the Atari 8-bit computers. Some are great, some are terrible, and most are at least adequate. True, most of these books are hard to find. Three years ago, the bookshelves had a handful of books about dozens of different kinds of computers. Now, instead, we find dozens of books about a handful of computers. Still, your bookstore can usually order what you need. And if it can't, try an Atari dealer. If that doesn't work, try one of the bigger mail order places that specializes in Atari.

Anyway, here's my point: If you think information about the 8-bit line is sparse, wait until you try to find out anything about the 16-bit machines! As I write this, the only book published so far is called *Presenting the Atari ST*. But don't expect to learn much from it that isn't in Atari's own somewhat skimpy (though attractive) manual. Yes, I have heard of additional books that are "in the works." But how long do you think it will be before there are 60 or 70 titles?

So I'm asking: "Why buy one of the new machines? Why not buy an 800XL or 130XE?" On the basis

of price alone, the 8-bit machines win handily. Atari recently announced a special promotion: 130XE, 1050 disk drive, 1027 printer, *AtariWriter*, and DOS 2.5 for \$399. Use your TV for the video, throw in a better programming language or business package and a game or two, and you're ready to enjoy computing for about five bills. Try to do the same thing with a 520ST, and you're going to spend about \$1,300 to \$1,400, presuming you want a color monitor. For an equivalent Amiga, add about \$800. What does this extra money buy?

Theory Versus Practice

In theory, the 16-bit machines should run programs 4 to 20 times faster than the 8-bit beasts. In truth, speed depends on the language and how well it is implemented. ST Logo is generally no faster than 8-bit Atari Logo. And for anything except possibly heavy math and intensive disk operations, neither Amiga's ABASIC nor ST BASIC are significantly (i.e., more than 25 percent or so) faster than OSS BASIC XE running on an XL or XE computer.

How about the theories that the new machines can run larger programs, display better graphics, use mouse control, and so on? As I write this, those are mostly just theories, waiting for people to write software and prove them. I have often told people contemplating the purchase of a computer that they should seek out a piece of software to fulfill their needs first, and only then ask what machine(s) it runs on. I cannot emphasize that advice enough for these new computers.

Does this mean that I think everyone should buy 8-bit machines and forget the new ones? Not at all! I simply question whether most people can benefit from their as-yet unrealized potential. And even when their power finally

arrives, how many home users will need more than what they get with, say, a 130XE? Business, scientific, and other users may very well need the extra speed and power, but it's pretty hard to justify an extra \$500 to \$1,500 if all you do with your computer is write a few letters a month and balance your checkbook.

What about people who want to learn how to program? They are total novices on computers, but enthusiasm is a great emptier of the pocketbook. Aside from the fact that there are lots of books on learning how to program an 800XL or 130XE, and none on how to use an ST or Amiga, how hard is it to learn to program on these new wonder machines? Well, writing plain-vanilla BASIC programs without graphics is reasonably easy. But that's easy on the XL and XE machines, also. Simple graphics, with lines and colors? Easy on both kinds of machines. Moving objects? Now we are getting to where it depends on the language: very easy with Atari 8-bit Logo, BASIC XE, and Amiga ABASIC; nearly impossible for a beginner with Atari BASIC or ST BASIC.

I guess I've made my points. As for me, I am moving on to the 16-bit machines. I am ready to learn new and different things, such as how artificial intelligence programs work. Such as how to manipulate multiple screen windows when writing a business application in Pascal. Such as...well, you get the idea. But I still enjoy programming in BASIC. And I still have a library of dozens of programs (mostly public domain and therefore free, or nearly so) which I enjoy on my 130XE. So I won't abandon any of you soon. As for yourself, think hard and read a lot before you abandon your trusty 8-bitter. ©



Odd Facets Of GEM

This month we're going to explore a handful of quirks in the GEM desktop: things you can do to make your system more useful...things you can do to make your system crash! Since I'm a natural-born pessimist, let's start with the crash.

The bug I'm about to demonstrate infests the ROM (Read Only Memory) version of the TOS operating system. Even so, if you don't already have the TOS ROMs, get them today. The difference in overall system performance and capability is only a little short of great.

To see this bug, simply boot your system and bring up the Control Panel. Select either the date or time field. Then type an underline character (SHIFT-hyphen). Watch your system bomb. The only recovery is to press the reset button or turn off the power.

Problem: The Control Panel is a form of dialog box, and it uses what are known as *editable text fields* to display and let you modify the date and time. An editable text field is designed to restrict the user to typing certain characters. For example, the date and time fields of the Control Panel are editable fields which allow only numbers to be typed. Unfortunately, somehow a bug crept into the ROM-based TOS. Anytime you edit a numeric-only field, typing the underline causes something nasty to occur. Editable fields for filenames have a similar, though usually nonfatal, problem.

Solution: A GEM application program that needs to accept numeric-only input from the user has two choices: (1) Use an editable field which allows *any* character and then validate the user's input after the dialog box returns; (2) Retrieve keystrokes one at a time, checking them on the fly, and print only the valid ones on the screen. The former solution is kind of ugly

because the user doesn't get immediate response to incorrect input. The latter solution is a lot of work. Take your pick.

Modifying DESKTOP.INF

Many of you already know how to customize the GEM desktop so your preferences appear automatically when you boot up the ST. When you select the *Save Desktop* item under the *Options* menu, GEM saves a file to the disk in drive A called DESKTOP.INF which stores these preferences. You can rearrange the icons on the screen, change screen colors, resize the windows, and so on, and GEM remembers it all for you.

DESKTOP.INF is an ordinary ASCII file, so it can be modified with most text editors and word processors. This lets you personalize GEM even more. (See "ST Hints & Tips," COMPUTE!, June 1986.)

The first thing we'll do is the easiest. Using a text editor or word processor that handles ASCII files, load and examine DESKTOP.INF. You should see one or two lines which contain the words FLOPPY DISK (among other things). These are the labels which appear beneath the disk icons. I usually rename the labels —*Top*—*Disk*— and *Bottom*—*Disk*. (I've used dashes here to show where I typed a space—magazine typesetting sometimes makes it hard to indicate spaces.)

Save the modified file back on disk in ASCII format. The next time you boot from that disk, the names should appear as you have modified them. Just for fun, sometimes I change the name of the trash can to *Junk!* or *Garbage* or something equally silly.

Rearranging Files

There are even more interesting things you can do with DESKTOP.INF. If, like me, you have a

disk or subdirectory in which you do most of your work, you'll soon find that you can't see all of the filenames or icons on the screen at once. Although it's a minor nuisance, it always seems that the files (or, more likely, programs) which I want the most are always off the screen. How can we force them back on the screen? Preferably in the upper-left position?

One solution, since the default display mode under the *Show* menu is *Sort by Name*, is to name your favorite files AARDVARK.PRG or AAABASIC.PRG. But that's kind of messy. A better method might be to choose *Sort by Date* if you could change the file's creation date. But I think Mark Rose (of Optimized Systems Software) has hit upon the best scheme.

First, he chooses *Sort by Type*. Second, he renames his most-used programs so they have no type (filename extension) at all. Third, he loads DESKTOP.INF and adds a line or so. To figure out exactly what to add, look for a line in DESKTOP.INF similar to this:

```
#G 03 FF *.PRG@ @
```

This line tells the desktop that all files which match the *.PRG specifier are GEM (G) program files. Now, let's say the program you want to appear at the top-left of the screen was called PASCAL.PRG and has been renamed to simply PASCAL. You would add this line to the end of the DESKTOP.INF file:

```
#G 03 FF PASCAL.@ @
```

This tells GEM that PASCAL is actually a GEM-based program. Neat, huh? What's more, you can do this for several files. However, I *do not* recommend using the *. wildcard in such a line—general untyped files end up looking like programs, a dangerous practice. ©



Tried And True Tools

In keeping with COMPUTE!'s programming languages theme for this month, I'd like to share some thoughts about programming in general and better use of the available languages in particular. I have long contended that, for most purposes, owners of Atari 400/800, XL, and XE computers have all the languages they need. You won't do parallel array processing with a 6502, no matter what language you use, but you *can* balance your checkbook, keep track of your mailing list, access online services via modem, write a book or two, and (of course) play some games. All of those applications and many more have been written with languages now available for the eight-bit Atari computers. What more can you ask for?

In a previous column I said it would be hard for most users to justify trading up to an ST, an Amiga, or whatever. If anything, I feel more strongly about that now. I still write this column using a good old Atari 1200XL (I like its keyboard best) and an Atari 825 printer (ancient history). Sometimes I wish for an 80-column screen or a hard disk drive—keeping track of 200 floppies is not my idea of fun—but I can't justify the expense for the extra convenience.

The same is true when it comes to programming languages. Admittedly, I'm a language junkie. I love learning new languages and/or tricks with old languages. So it would seem that the ST would be a dream machine for me. Despite its youth, the number of languages either available or coming soon is phenomenal: several varieties of BASIC, Logo, Pascal, C, LISP, Modula-2, COBOL, FORTRAN, Prolog, Forth, and 68000 machine language. There are probably others, too.

Old Machines, New Projects

But for owners of eight-bit Ataris, the situation is far from bleak. Though some of the language implementations are not as rich as those on the ST, we can enjoy Pascal, C, Logo, Action!, Forth, PILOT, 6502 machine language, and some extraordinarily easy-to-use BASICs.

Even though I've been using Ataris for six years now, I still see some interesting projects to do—projects that I've never done or which I think can be done better. A few examples: How about a terminal program written in Action! that is designed to work well with CompuServe's conference mode? Or a GEM-like interface for DOS? Or a combined spreadsheet/database written in BASIC XE and commented liberally so that even beginners can see the methods used? I know I'll never do all of these, but they are challenges I'd *like* to tackle.

Rethinking The Problem

Moving to new languages on new machines is not always an advantage. For instance, in ST BASIC, strings cannot exceed 255 characters in length. Atari BASIC strings can be up to 32,767 characters long, if you have the memory available. (Yes, ST BASIC allows string arrays, but so do BASIC XL, BASIC XE, and Atari Microsoft BASIC.) There are many other examples.

Another factor is that the speed and power of the newer machines is of little advantage for some applications. Other than missing an 80-column screen, I can use CompuServe or various bulletin boards just as well with my \$100 computer as I can with the company's \$1,000 machine. Besides, the modem for the \$100 computer is cheaper. And by the time you read this, Atari may have released its 80-

column adapter for the eight-bit line.

Suppose you're writing a program that *does* need more speed, however. What can you do other than buy a newer, faster computer? Well, you could buy a better, faster language. That's a lot cheaper than buying a new computer—for which you still might need an extra language or two. On the other hand, maybe you don't have to buy anything at all; maybe you just need to rethink your solution to the problem. Let me show you what I mean.

Program 1 is very similar to one which I found in a recent user group newsletter. The author was responding to a member's inquiry about writing a routine to shuffle a deck of cards. As you know, when BASIC gives you a random number, there's no guarantee it won't give you that same random number twice, perhaps even several times. For a quick example, type the following line and press RETURN:

```
FOR I=0 TO 9:PRINT INT(10*RND  
(0)):NEXT I
```

This asks for ten random numbers in the range 0 to 9. Did you actually get ten different numbers? The odds are very much against it.

The Super Shuffle

Program 1 demonstrates this problem by dealing out an entire deck of 52 cards. As each card is dealt (by suit S and rank R, line 210), its spot in the C (card) array is marked. Then, if the random number generator picks that card again, the pick is ignored (line 230). The only things I added to the original routine are the counters (C and T) which count how many picks it takes to get each card and the entire deck. Can you guess how many picks it takes to get the entire deck? In 50 tests, it took a minimum of 128 picks and a maximum of 457, with the average around 220. The

Now look at Program 2, which does exactly the same job but never takes more than one pick to get the next card in the deck. It works by using a single string (CARD\$) to represent the entire deck. When it gets a random number from 1 to 52, the program removes the corresponding "card" from the "deck" (lines 400 and 410). The next time it picks a card, it gets a random number from 1 to 51. Each time the computer gets a card, the range of random numbers gets smaller. Simple. And it works by taking advantage of the string operations in Atari BASIC.

The point of this exercise is to show that sometimes the best way to fix a slow or inefficient program is to rethink it and then rewrite it. I'd be willing to bet that Program 2 *on an eight-bit Atari* runs faster than Program 1 on an Atari ST. If you have access to both machines, you might want to try it. And try improving your own programs. (Even while writing this column, I found a way to improve Program 2 even more. Can you find it?)

One last comment: Notice the readability of the two programs. Which one is cryptic and which one almost explains itself? Meaningful variable names can add a great deal of value to any program.

For instructions on entering these listings, please refer to "COMPUTE!'s Guide to Typing In Programs" in this issue of COMPUTE!.

Program 1: Slow Shuffle

```

MD 100 DIM C(4,13)
OP 110 DIM R$(13):R$="A23456
      789TJQK"
IS 120 DIM S$(4):S$="{,},,{,}
      (.)(P)"
AH 130 FOR S=1 TO 4:FOR R=1
      TO 13:C(S,R)=0
PC 140 NEXT R:NEXT S
CE 200 FOR I=1 TO 52:C=0
NC 210 S=INT(RND(0)*4)+1:R=I
      NT(RND(0)*13)+1
LD 220 C=C+1
PV 230 IF C(S,R)>0 THEN 210
BJ 240 C(S,R)=1
OK 250 T=T+C
LB 260 PRINT I;:POKE 85,15-L
      EN(STR$(C)):PRINT C;:
      " PICK(S) TO GET ";R$(
      R,R);" OF ";S$(S,S)
CB 270 NEXT I
PJ 280 PRINT :PRINT "TOTAL P
      INKS: ";T

```

Program 2: Fast Shuffle

```

ED 100 REM === SET UP VARIABLES, ETC. ===
BC 110 DIM CARD$(52)
BA 120 DIM SUITS$(8*4)
CA 130 SUITS$="SPADES HEARTS CLUBS{3 SPACES}DIAMONDS"
GB 140 DIM SUIT$(8)
PH 150 DIM RANKS$(4*5)
EE 160 RANKS$="ACE JACK KING QUEEN"
EI 170 DIM RANK$(5)
GN 200 REM === SET UP THE DECK ===
AM 210 FOR CARD=1 TO 52
OG 220 CARD$(CARD)=CHR$(CARD)
OD 230 NEXT CARD
IM 240 DECKSIZE=52
MP 300 REM === DEAL 52 CARDS ===
AN 310 FOR CARD=1 TO 52
HC 320 PICK=INT(DECKSIZE*RND(0))+1
LP 330 PICKED=ASC(CARD$(PICK))-1
JI 340 SUIT=INT(PICKED/13)
LB 350 RANK=PICKED-13*SUIT
IF 360 SUIT$=SUITS$(SUIT*8+1, SUIT*8+8)
HD 370 IF RANK<4 THEN RANK$=RANK$(RANK*5+1, RANK*5+5)
LK 380 IF RANK>=4 THEN RANK$=STR$(RANK-2)
DK 390 PRINT "Picked: "; RANK$; " OF "; SUIT$
AN 400 IF PICK<DECKSIZE THEN CARD$(PICK)=CARD$(PICK+1)
EA 410 IF PICK=DECKSIZE THEN CARD$(PICK)=""
NF 420 DECKSIZE=DECKSIZE-1
PA 430 NEXT CARD

```

Save Your Copies of COMPUTE!

Protect your back issues of *COMPUTE!* in durable binders or library cases. Each binder or case is custom-made in flag-blue binding with embossed white lettering. Each holds a year of *COMPUTE!*. Order several and keep your issues of *COMPUTE!* neatly organized for quick reference. (These binders make great gifts, too!)



Binders

<p>Case:</p> <p>\$8.50 each; 3 for \$24.75; 6 for \$48.00</p>	<p>Case:</p> <p>\$6.95 each; 3 for \$20.00; 6 for \$36.00</p>
--	--

Cases:

\$6.95 each;
3 for \$20.00;
6 for \$36.00

(Please add \$2.50 per unit
for orders outside the U.S.)

Send in your prepaid order
with the attached coupon

Mail to: Jesse Jones Industries
P.O. Box 5120
Dept. Code COTE
Philadelphia, PA 19141

Please send me _____ COMPUTE! ☐ cases ☐ binders.
Enclosed is my check or money order for \$ _____. (U.S. funds only.)

Name _____
Address _____
City _____
State _____ Zip _____

Satisfaction guaranteed or money refunded.
Please allow 4-6 weeks for delivery.

To receive additional information from advertisers in this issue, use the handy reader service cards in the back of the magazine.



Your Roving Reporter

I attended (and exhibited at) the Los Angeles Atari Computer Faire on August 15 and 16, and I would like to share a few things I saw and a few thoughts I had. The most significant part of this Faire was probably Atari's presence. As far as I know, this was their first real participation in a user group-sponsored event, and they were there in force. Such Atari notables as Sam Tramiel, Sig Hartmann, John Feagans, Neil Harris, Mel Stevens, and Sandy Austin (and others who will undoubtedly embarrass me by asking me why I forgot them) all made an appearance.

Faster ST Graphics

Most important were the products being shown there for the first time. Atari's new blitter chip for the ST was being put through its paces. This chip takes over some of the graphics processing (such as moving sprites) that must be done in software on current ST machines. Depending on the type of processing, this chip should make graphics-oriented programs run from two to six (my estimate) times faster.

The 80-column adapter for the eight-bit machines was also on display, as was a new word processor for the ST: *Microsoft Write*. Although it is another nice, solid word processor, I did not see any really exciting features. But the very presence of Microsoft in the Atari world is expected by many to lend respectability to the ST machines.

The real battle for attention, though, was among the various purveyors of music software for the ST machines, particularly by the MIDI-oriented companies. Sounds ranged from exotically electronic to a guitar so realistic I thought it was a live accompaniment.

The Catalog (honest, that's the company's official name) people were showing off animated 3-D

graphics, which wasn't too surprising, given the capabilities of Tom Hudson's *CAD 3-D* program. But then they added liquid crystal "shutter" glasses for *true* 3-D vision and a glimpse into some fascinating future possibilities. Liquid crystal glasses are not exactly a convenience store item (they usually go for hundreds or even thousands of dollars—mostly to the military), but you can expect to buy a pair sometime early next year for \$150 or so, according to the exhibitors.

Significantly missing, though: the game companies. No Brøderbund, no Sierra On-Line, and so on. What a turnaround from the early days of the West Coast Computer Faire. Most attendees probably didn't complain, though, since there was a good deal of software for eight-bit and ST machines. There were literally hundreds of titles available in each category, even though the Faire organizers purposely limited the number of dealers at the Faire to four, and one of those sold no eight-bit software.

Finally, the show was put on by an association of user groups, and almost every member I talked to was pleased by the show and the turnout. Final figures were not in as I left, but John Tarpanian, president of both HACKS and ACENET, estimated the crowd for the two-day event at 3000 people. It seemed at least that big. Atari is encouraging at least two more such shows that I know of: one right here in San Jose in September, and one in Portland in October. There's another show in Virginia in November, though it's not as closely tied to Atari as these other three. I suggest attending one of these if you can.

Join Your Local User Group

This is the first of my answers to readers' inquiries, and it ties in

neatly with the discussion above. Several people asked me where they could get (1) help with their hardware and/or software, (2) cheap public-domain programs, or (3) up-to-date news on events of the Atari world. My answer to all three? *Join a user group.*

I have pushed user groups in this column before, and I will probably do so again. At the Faire, for example, one person thanked me for getting him involved in a group—he had quickly gotten the help he needed. I asked him if he's now returning the favor to newer members. He is. He's the club's librarian.

There are over 300 active user groups in the U.S. now, so there's a good chance there is one near you. And if you join one, maybe you can help put on one of these Faires in the next year or two.

How do you find a user group? Ask a local dealer or look for announcements in newspapers. And user groups: Be sure to have a publicity chairperson who gets you mentioned in your hometown paper from time to time. If you are absolutely desperate, send me a *self-addressed, stamped envelope*, and I will give you the address of the closest group on my list. Send your request to P.O. Box 710352, San Jose, CA, 95171-0352. No guarantees of a good match, though.

Also, if you have a modem, you might like to know that I have an account on CompuServe. You can leave messages for me by using my account number: 73177,2714. I expect to be active on Delphi in the near future, too, but I don't have an account name for that service yet. Please understand that I cannot give long-winded answers online. It costs money, remember. But I don't mind comments, suggestions, or even quick questions. ©



Number-Base Conversions

This column was prompted by a letter in *COMPUTE!*'s letters column, in which the author asked for a program to convert decimal numbers to binary. "Why," I asked myself, "do all these conversion programs work with only one pair of bases (for example, base 10 to base 2)?" Answer: because few realize that a more general program is almost as easy these specific ones. Don't believe me? Keep reading.

Number Bases

You probably learned about number bases back in third or fourth grade, though you might not have realized that's what you were learning. Specifically, you likely were taught that the number 735 represented "seven hundreds, three tens, and five ones." The fact that digits in a number represent powers of ten is kind of an accident. If humans were normally born with only three fingers and a thumb on each hand, you can bet that 735 would have meant "seven sixty-fours, three eights, and five ones" (that is, we would have used base 8).

Since computers are "born" with only two "fingers," their natural tendency is to use base 2, also known as *binary* numbers or notation. (A computer's "fingers" are its memory cells, but each cell can remember only off or on, equivalent in function to counting on two human fingers.) Yet you seldom see a computer memory dump printed in binary, simply because such a printout would be gigantic! Binary numbers take up a lot of room compared to equivalent decimal numbers. Instead, because of the neat way that powers of two can be grouped together, we tend to see computer memory represented in either octal (base 8) or hexadecimal (base 16) notation.

One thing you may have noticed is that a base's number is the

same as the number of counting symbols needed to represent it. Thus base 10 uses 0-9. Base 8 uses only 0-7. What about bases beyond 10, such as base 16, the hexadecimal base most often used in microcomputer work? Doesn't it need 16 counting symbols? Yes, indeed, and the symbols most commonly used are 0-9, followed by A-F. (Why not use completely new symbols for the digits beyond 9? Simple: Early computer printers had only 64 different symbols available, so uppercase letters were used.)

Why Hex?

Sidelight: Since we are working on computers that tend to work with bytes, and since a byte can have a value from 0 to 255 (decimal), base 256 notation would seem to be a logical choice. But now we can see why it is not used—humans would be forced to learn 256 unique digit symbols! Still, there are two "nybbles" in each byte, and a nybble can have a value from 0 to 15 (decimal), so hexadecimal (base 16) notation is a very logical alternative.

Now, when you see a hexadecimal number such as A88C, what does it mean? Well, you can read that as "A four-thousand-ninety-sixes, 8 two-hundred-fifty-sixes, 8 sixteens, and C ones." In turn though, A and C may be read in decimal as 10 and 12, respectively. Whew! Now how about base 19?

Confused? Don't worry, help is at hand. Program 1 consists of a short main program followed by two special-purpose subroutines. These routines are designed to make it easy to allow entry and display of any number using any base or pair of bases. The first one (from line 9200 to line 9330) takes a number in variable N and converts it to a string in variable N\$ using the number base given by the variable BASE. The second routine

(lines 9400-9560) performs the reverse operation, converting a string in N\$ (which is supposed to be a number in BASE notation) and converting it to N for use as a number anywhere in BASIC.

Try it. Type in the main code and the subroutines and try the various options. And use some bizarre number bases, such as 13 or 37 or 53. In keeping with the tradition of hexadecimal numbering, the digit symbols used are 0-9 (same as decimal for the first ten symbols), followed by A-Z, and then a-z (good enough for anything up to base 62!).

So now I have one set of routines which take care of *all* conversions. And it's kind of fun. You could even make a game of it: Try to make two English words "equal" by changing bases! For example, RIB base 35 equals some animal (which happens to enjoy ribs) in some other base. Can you find the animal word and its base? Maybe tricks like this could make a hard-to-break encryption scheme? (This can really cause you to lose sleep!)

Be Just A Bit Wiser

I couldn't quit with simple number conversions, of course. One of the handy features of most higher-level languages is (usually) the presence of operators which do *bitwise* operations. I like such operators so much I put them into the first of the advanced Atari-compatible BASICs we did, way back in 1981. Unfortunately, Atari BASIC does not have bitwise operators. In Atari BASIC, operators such as AND and OR always perform logical comparisons rather than bitwise comparisons. Though, in fairness, I should point out that there are occasions where Atari's logical operators are worth as much as or more than bitwise operators. Some authors have agreed with me to the extent that they have written machine

language USR calls for use in their BASIC programs. But this is beyond the ken of most BASIC users.

Fortunately, bitwise operators can be implemented in Atari BASIC. And that's exactly the purpose of the subroutines of lines 9000 through 9090 (bitwise AND) and lines 9100 through 9190 (bitwise OR) in Program 2. I don't have space in this column to explain the theory and operation of bitwise operators, but we can quickly look at one example of their use.

Suppose you want to perform some subroutine only when the user of your program hits the SELECT key. Further, suppose that in your program it is legitimate and possible that the user may be pushing down either (or both) the START and OPTION keys at that same time as SELECT. If you look in most any good reference book (COMPUTE!'s *Mapping the Atari*, for example), you will find a little table something like this:

Push this key	PEEK(53279) decimal	shows this binary
none	7	111
START	6	110
SELECT	5	101
OPTION	3	011

Here we have listed the binary values (even though you could have run Program 1 to convert the decimal values yourself) to show clearly what the console keys are doing: Each of those three keys changes a single bit of the specified address from 1 to 0 when it is pushed. So, we would like a way to isolate the state of the middle bit (of the three) to test for SELECT being pressed. No sooner said than done.

In most languages, you would use something equivalent to this:
SELECTPUSHED = NOT (PEEK(53279) AND 2)

In Atari BASIC, you can do it the way I did it in Program 2. Enough said?

Finally, there is Program 3. You can *not* use this program by itself. You must first add all four of the subroutines (on lines numbered 9000 or greater) from Programs 1 and 2. Be sure to keep those subroutines handy so they can be used by Program 3 or, I hope, by some of your own programs. (Remember, if you LIST a range of lines to disk or cassette, you

can use ENTER to merge them with a program in memory.)

Program 3 is a catchall. It allows you to enter two numbers using two (optionally) different number bases. It then allows you to choose a number base for display purposes and shows you the conversions of the two numbers along with the results of bitwise ANDing them and bitwise ORing them. For a thorough understanding of bitwise operations, you might choose base 2 (binary) for all input and output. Happy hacking. ©

Program 1: Base Converter

```

EI 100 REM ***** PROGRAM TO
      DEMONSTRATE
DC 110 REM ***** NUMBER BASE
      CONVERSION
HM 120 REM
EH 130 DIM N$(40):REM (MUST
      BE AT LEAST 32)
HJ 140 REM
JD 200 ? :? :PRINT "BASE FOR
      INPUT";:INPUT BASEIN
KE 210 PRINT "NUMBER ";:INP
      UT N$
NJ 220 BASE=BASEIN:GOSUB 940
      0:DECIMAL=N
CE 230 IF N<0 THEN PRINT "OO
      PS":GOTO 200
HK 240 PRINT "BASE FOR OUTPU
      T";:INPUT BASE
CE 250 PRINT
NK 260 PRINT N$;" BASE ";BAS
      EIN;" = "
NJ 270 GOSUB 9200:PRINT N$;"
      BASE ";BASE;" = "
BB 280 PRINT DECIMAL;" BASE
      10"
BB 290 GOTO 200
BP 9200 REM ***** CONVERT N
      TO N$ USING GIVEN BA
      SE
PN 9210 REM ENTER: N,BASE
JB 9220 REM USES: (3 SPACES)D
      IG$,DIGIT,WORK,TEMP
NF 9230 TRAP 9250
FK 9240 DIM DIG$(62)
FE 9250 DIG$="0123456789ABCD
      EFGHIJKLMNOPQRSTUVWXYZ
      abcdefghijklmnopqr
      stuvwxyz"
ME 9260 N$="0000000000000000
      0000000000000000"
KA 9270 WORK=N
DL 9280 FOR DIGIT=32 TO 1 ST
      EP -1
LF 9290 TEMP=INT(WORK/BASE):
      WORK=WORK-TEMP*BASE
MC 9300 N$(DIGIT,DIGIT)=DIG$
      (WORK+1)
GO 9310 WORK=TEMP:IF WORK TH
      EN NEXT DIGIT
HD 9320 IF N$(1,1)="0" THEN
      N$=N$(2):GOTO 9320
KP 9330 RETURN
CB 9400 REM ***** CONVERT N$
      TO N USING GIVEN BA
      SE
II 9410 REM ENTER: N$ HAS P
      RESUMED NUMBER IN ST
      RING FORM

```

```

ED 9420 REM . (7 SPACES)BASE
      IS BASE TO USE
BJ 9430 REM EXIT: (3 SPACES)N
      HAS NUMBER IN INTER
      NAL FORM (<0 IF ERRO
      R)
AC 9440 REM USES: (3 SPACES)D
      IGIT,TEMP
QJ 9450 REM NOTE: (3 SPACES)D
      IGBITS GO TO BASE 66,
      IN ORDER
PA 9460 REM . (7 SPACES)0..9,
      A..Z,A..Z
HP 9470 IF N$(1,1)="0" THEN
      N$=N$(2):GOTO 9470
JA 9480 N=0
OB 9490 FOR DIGIT=1 TO LEN(N
      $)
ND 9500 TEMP=ASC(N$(DIGIT))-
      48:IF TEMP<0 THEN N=
      -1:RETURN
BF 9510 IF TEMP>9 THEN TEMP=
      TEMP-7:IF TEMP<10 TH
      EN N=-1:RETURN
EH 9520 IF TEMP>35 THEN TEMP
      =TEMP-6:IF TEMP<36 T
      HEN N=-1:RETURN
FO 9530 IF TEMP>=BASE THEN N
      =-1:RETURN
FB 9540 N=N*BASE+TEMP
ID 9550 NEXT DIGIT
LE 9560 RETURN

```

Program 2: Bitwise Operations

```

ME 100 REM ***** PROGRAM TO
      SHOW STATE
PP 110 REM ***** OF CONSOLE
      KEYS AND
HJ 120 REM ***** DEMONSTRATE
      BITWISE AND
AB 130 X=PEEK(53279):IF X=7
      THEN 130
NP 140 Y=1:GOSUB 9000
FE 150 IF NOT XANDY THEN PR
      INT "START",
OC 160 Y=2:GOSUB 9000
II 170 IF NOT XANDY THEN PR
      INT "SELECT",
OB 180 Y=4:GOSUB 9000
KD 190 IF NOT XANDY THEN PR
      INT "OPTION",
BP 200 PRINT
PJ 210 IF PEEK(53279)=X THEN
      210
FO 220 GOTO 100
CD 9000 REM **** REM BITWISE
      AND
ED 9010 REM ENTER: X,Y
MH 9020 REM EXIT: (3 SPACES)X
      ANDY IS X AND Y
ML 9030 REM USES: (3 SPACES)T
      EMPX,TEMPY,MASK
EO 9040 TEMPX=X:TEMPY=Y:XAND
      Y=0:MASK=1
PG 9050 TEMPX=INT(TEMPX)/2:T
      EMPY=INT(TEMPY)/2
AF 9060 IF TEMPX=0 OR TEMPY=
      0 THEN RETURN
KD 9070 IF TEMPX<>INT(TEMPX)
      AND TEMPY<>INT(TEMP
      Y) THEN XANDY=XANDY+
      MASK
LN 9080 MASK=MASK+MASK
NJ 9090 GOTO 9050
AO 9100 REM **** BITWISE OR
EE 9110 REM ENTER: X,Y
GE 9120 REM EXIT: (3 SPACES)X
      ORY IS X OR Y
MH 9130 REM USES: (3 SPACES)T

```



```

EMPX,TEMPY,MASK
BN 9140 TEMPX=X:TEMPY=Y:XORY=0:MASK=1
PH 9150 TEMPX=INT(TEMPX)/2:TEMPY=INT(TEMP
Y)/2
DI 9160 IF TEMPX=0 AND TEMPY=0 THEN RETUR
N
AO 9170 IF TEMPX<>INT(TEMPX) OR TEMPY<>IN
T(TEMPY) THEN XORY=XORY+MASK
LO 9180 MASK=MASK+MASK
NL 9190 GOTO 9150

```

Program 3: Subroutine Demo

```

EI 100 REM ***** PROGRAM TO DEMONSTRATE
DC 110 REM ***** NUMBER BASE CONVERSION
BP 120 REM ***** AND BIT-WISE OPERATORS
HI 130 REM
EI 140 DIM N$(40):REM (MUST BE AT LEAST 3
2)
HK 150 REM
CH 200 ? :? :PRINT "IN BASE ";:INPUT BASE
KE 210 PRINT "NUMBER ";:INPUT N$
PO 220 GOSUB 9400:X=N
CE 230 IF N<0 THEN PRINT "OOPS":GOTO 200
BJ 240 PRINT "BASE FOR INPUT":INPUT BASE
KI 250 PRINT "NUMBER ";:INPUT N$
AD 260 GOSUB 9400:Y=N
CI 270 IF N<0 THEN PRINT "OOPS":GOTO 200
HO 280 PRINT "BASE FOR OUTPUT":INPUT BAS
E
CI 290 PRINT
PI 300 PRINT "----- RESULTS -----"
DP 310 N=X:GOSUB 9200:PRINT "FIRST NUMBER
":N$
HF 320 N=Y:GOSUB 9200:PRINT "SECOND NUMBE
R ":N$
GD 330 GOSUB 9000:GOSUB 9100
AH 340 N=XANDY:GOSUB 9200:PRINT " BITWIS
E AND ":N$
KE 350 N=XORY:GOSUB 9200:PRINT " BITWISE
OR ":N$
GE 360 GOTO 200
NP 999 REM ***** END OF MAIN CODE ***** ©

```

Public Domain & User Supported Software

NEW TOP TEN FOR COMMODORE 64

- The 64 GOLD Library **\$5.00/DISK**
- ☐ 105 ARTIST SKETCHBOOK drawing programs
 - ☐ 106 GREAT AMERICAN NOVELISTS word processing
 - ☐ 107 PHONE CONNECTIONS communications
 - ☐ 108 SPACE WARS space games
 - ☐ 109 DUNGEONS & DRAGONS text adventures
 - ☐ 110 HOME ORCHESTRA instrument simulation
 - ☐ 111 JUKE BOX prerecorded songs
 - ☐ 112 EINSTEIN'S FAVORITES advanced math
 - ☐ 113 PONZO'S TUTOR programming from BASIC to machine
 - ☐ 114 ELECTRONIC SECRETARY filehandling utilities

NEW TOP TEN FOR IBM \$6.00/DISK

- PC-SIG Authorized Dealer
- ☐ 005 PC-FILE III, V4 labels, forms, and more
 - ☐ 078 PC-WRITE v2.165 popular and powerful
 - ☐ 273 BEST UTILITIES print spooler, file search, more
 - ☐ 274 BEST GAMES packman, breakout, wizard, more
 - ☐ 293 ARCADE GAMES (color graphics required)
 - ☐ 405 DESKMATE more than a sidekick
 - ☐ 457 GREATEST ARCADE the best of the best games
 - ☐ 528 NEW YORK WORD sophisticated word processing; 1 of 2
 - ☐ 529 NEW YORK WORD 2 of 2
 - ☐ 557 PINBALL ALLEY from simple to complex pinball games

NEW TOP TEN FOR APPLE \$5.00/DISK

- ☐ 037 FREEWRITER wordprocessor (Apple II + needs paddles)
- ☐ 038 BUSINESS/HOME MANAGEMENT checkbook, calculator, more
- ☐ 039 BEST OF BUSINESS general ledger, payroll, much more
- ☐ 056 BANK'n SYSTEM check balancer, write & print checks
- ☐ 057 OMNI FILE data base with instructions
- ☐ 064 BEST OF EDUCATION math drills, spelling, typing, etc.
- ☐ 085 BASIC MATH DRILLS fractions, multiple choice, work problems
- ☐ 118 GAMES fast action space arcade games
- ☐ 195 PASSIONS nude women, game
- ☐ 213 BEST UTILITIES diskcat, krunch, diskcheck, diskmap, etc.

NEW TOP TEN FOR MAC \$9.00/DISK

- ☐ 005 CODE CRACKING, FEDIT edit file blocks in ASCII or hex
- ☐ 006 ReSED and ReED edit menu bars, icons and I.D. numbers
- ☐ 007 SWITCHER edit multiple Microsoft BASIC files
- ☐ 029 COMMUNICATIONS Red Ryder, MacPep
- ☐ 037 SLIDE SHOW
- ☐ 039 FONTS Font catalog
- ☐ 045 DESK ACCESSORIES Minifinder, timer
- ☐ 062 GAMES Dungeons of doom, baseball
- ☐ 067 GAMES Billiards, volleyball, juggling
- ☐ 086 BEST OF MAC MacWorld 86

PUBLIC DOMAIN SOFTWARE EXCHANGE
Authorized Dealer

Add \$4 shipping & handling per order. CA residents
add 6.5% sales tax
Amount enclosed \$ _____ ☐ Check ☐ VISA ☐ MasterCard

Card No. _____
Signature _____ Exp. Date _____
Phone (_____) _____
Name _____
Address _____
City _____ State _____ Zip _____

Call toll free 800-431-6249
in Calif. 415-952-1994



BLACKSHIP
COMPUTER SUPPLY
P.O. Box 883362
San Francisco, CA 94188

CAPUTE!

Atari DOS Switcher

This utility program from the December 1986 issue (p. 71) does not work as published because the final five lines of the listing are missing. To create a working version, add the following lines:

```

NA 1140 DATA 238,107,23,238,
111,23,238,114,23,23
8
KA 1150 DATA 118,23,173,107,
23,201,52,208,218,76
LG 1160 DATA 66,25,63,25,82,
25,76,70,23,173
JJ 1170 DATA 1,211,9,1,141,1
,211,169,64,141
LN 1180 DATA 14,212,88,160,1
,96,0

```

Fontier 128

Line 1890 of this program from the December 1986 issue (p. 82) cannot be typed in as listed. Unlike the Commodore 64, the 128 always

prints the key definitions rather than characters when the function keys are pressed. Replace the line with the following:

```

HD 1890 K$="+-{}[UP]{DOWN}{LEFT}
{RIGHT}"+CHR$(137)+CHR
$(139)+CHR$(133)+CHR$(
135)+CHR$(138)+CHR$(14
0)+CHR$(134)+CHR$(136)

```

Laser Strike

Line 700 of the Apple version of this game from the December 1986 issue (Program 2, p. 52) ends with an incomplete statement—there is nothing following the THEN. The IF statement was never executed during our extensive testing of the game, so this should not cause a problem. However, you may want to delete the bad statement. Remove everything in that line following (and including) the last colon.

Biker Dave For Atari

The corrections in last month's CAPUTE! column fix the bugs in the Atari version of this program from the November 1986 issue. However, we have discovered that the program will not function properly if you stop the program with the BREAK key and then restart it with RUN. If you ever have cause to break out of this program, you'll need to reset the computer and reload the program before continuing.

Lumpies Of Lotis IV

A comma was inadvertently removed during printing in line 260 of this IBM game from the October 1986 issue (p. 53). The line should begin with IF ABS(Z(X,Y,LEV-1)).

©



Corrected File Conversions

Well, this month marks a historic occasion for those of us at Optimized Systems Software. March 1981 was the month we introduced our first Atari-oriented products: BASIC A+, EASMD, and OS/A+ (called CP/A until a lawyer for DRI objected—maybe we could have fought them if we had had more than \$2.98 in our checking account). We finished those products off in record time and presented them at the West Coast Computer Faire. We managed to sell 17 (yes, that is 3 less than 20) packages at about \$120 each (that was cheap in those days), and we decided then and there we could stay in business for another month (maybe even two).

Well, the months kept passing like that. OSS has never been a wildly successful company—selling languages for a computer on which fewer than 10 percent of all owners actively program is not conducive to instant wealth—but we have always had some loyal followers. As I have mentioned here before, I started writing this column because I saw some questions in COMPUTE! about Atari software internals that I thought needed some answers. But I wouldn't have even gotten interested in reading COMPUTE! if we hadn't started OSS. See? All things are related when you look deep enough.

Unified We Stand

Speaking of software internals and answers.... In the recent issues of COMPUTE! there are a pair of programs which purport to convert standard Atari binary object files into either strings ("Stringing Atari Machine Language," September 1986) or DATA statements ("ML Write for Atari," January 1987). Both of these programs have a common limitation which was not mentioned in the articles accompanying

them: You *must* use them *only* with single-segment binary files. How do you know if a particular binary file consists of only a single segment? Glad you asked.

The program which accompanies this article is a simple little utility that analyzes any standard Atari binary file, printing the first and last address of each segment as it goes. When the program asks for the complete file name, you should enter the name of a binary file, including the disk drive specifier and extension (for example, D1:RAMDISK.COM). Watch the resultant screen display. If addresses for more than one file segment are displayed, then you may *not* use the programs described in those articles for this file.

Exception: If the addresses are *all* contiguous (that is, if the starting address of a segment is exactly one more than the ending address of the prior segment and if this holds true for all segments), you can use this file *if* you unify it first. I discussed segmented files in my April 1986 column and presented a unifying program there. Unfortunately, the program accompanying that article was misprinted, so you have to look in the article titled "Custom Characters for Atari SpeedScript" by Charles Brannon in the May 1986 issue (pages 88–90) for a corrected version of the file unifier.

If you are not comfortable with the hex addresses printed by the segment-checking program, you may view decimal addresses instead by replacing lines 110 through 150 below with just this one line:

```
110 PRINT "SEGMENT: ";START;"
      THROUGH ";QUIT
```

And one last caution: Though not mentioned in the article, machine language code placed in strings (as in the September 1986 article) *must* be intrinsically relocatable. Many of the routines floating

around on BBSs and in user-group libraries are indeed relocatable, but don't rely on this always being so. Test these routines in strings (or *any* machine language routines, for that matter) only *after* you have made sure you have saved your program and after you have put a junk diskette in the drive. (If you have an Indus drive or other drive that you can protect from the front panel, setting the protection is another adequate safeguard.)

Binary File Segment Checker

```

FI 10 REM **** BINARY FILE S
      EGMENT CHECKER ***
IG 20 DIM FILE$(20),HEX$(16)
      :HEX$="0123456789ABCDE
      F"
OE 30 GRAPHICS 0
BI 40 PRINT "COMPLETE FILE N
      AME";:INPUT FILE$
BH 50 OPEN #1,4,0,FILE$
OB 60 TRAP 200:GET #1,LOW:GE
      T #1,HI
KK 70 IF HI=255 AND LOW=255
      THEN GET #1,LOW:GET #1
      ,HI
KI 80 START=LOW+256*HI
EL 90 TRAP 40000:GET #1,LOW:
      GET #1,HI
IB 100 QUIT=LOW+256*HI
AL 110 PRINT "FILE SEGMENT:
      ";
JC 120 HEX=START:GOSUB 230
NB 130 PRINT " THROUGH ";
EJ 140 HEX=QUIT:GOSUB 230
CD 150 PRINT
EK 160 FOR ADDR=START TO QUI
      T
DA 170 GET #1,JUNK
PD 180 NEXT ADDR
DJ 190 GOTO 60
JP 200 REM *** GET HERE ON E
      ND OF FILE ***
DN 210 IF PEEK(195)<>136 THE
      N PRINT "UNEXPECTED E
      RROR # ";PEEK(195)
SL 220 END
PH 230 REM *** HEXPRINT SUBR
      OUTINE ***
IJ 240 DIV=4096
BE 250 FOR DIGIT=1 TO 4:TEMP
      =INT(HEX/DIV)
AK 260 PRINT HEX$(TEMP+1,TEM
      P+1);
OD 270 HEX=HEX-DIV*TEMP:DIV=
      DIV/16
EK 280 NEXT DIGIT
HL 290 RETURN
```



Retry, Retry, Retry Again

In the past, I have claimed in this column that I have *never* had an Atari disk "crash" on me if I wrote to it with verify turned on. Thanks to the little display panel on my Indus drives, I finally figured out why. A couple of times now, I have been writing programs or articles to disk (with verify turned on) and have noticed the Indus drive indicate that an error occurred. Yet neither DOS nor BASIC (nor the word processor) have reported any such error. Why?

Simple: Because of a safety feature in Atari DOS called a *retry counter*. Any time a disk operation fails, Atari DOS tries to perform the task again. In fact, it tries up to three more times. Now, if you write to the disk *without* verify, the drive may not know if something goes wrong (for example, if there is a defective spot on the disk). So you don't find out that the write failed until the next day (or week or year), when you try to reread the data. Kablooeey! Even the fact that DOS retries the read three times may not help.

On the other hand, if you write with verify turned on, the drive writes the information to the disk and then immediately reads it to be sure the write was successful. Assuming that there was an error, though, you would seem to be no better off than you were without verify mode turned on *except* that the drive tells DOS about the verify error and DOS tries again (and again and again if it needs to) to write that same sector.

Soft Errors

Usually, disk write errors are what are called *soft* errors. That is, if you try again, the write succeeds. And after three attempts, if the write still fails, that disk is in really bad shape (or you forgot to format the disk or remove the write-protect tab). Well, very few of my disks are in *really*

bad shape, so DOS tries again—and I don't even realize that I had a near miss with a blown file.

The point of all this is twofold: First, I would like to try to convince anyone doing work which would require a long time to reconstruct to use the "write with verify" mode. (If you are using DOS 2.5, this is easy, since the SETUP.COM program allows you to choose verify mode with just a few keystrokes. DOS 2.0 users can type POKE 1913,87 and then make the change permanent by using the DOS menu's WRITE DOS option.) Second, although three retries seem like enough for any write operation, I have had a few disks where a file seemed "slightly" damaged—that is, one time it would seem to read okay and the next time it would give me an error (usually ERROR 164). Well, I really *should* move that file to another disk (and I do), but how can I increase my chances that I can read it well enough to do the transfer? Answer: Change the number of retries that DOS makes.

Time for another program: Program 1 is a short BASIC program that you can use to modify DOS 2.5's retry count (and, incidentally, turn on verify mode for writes). Once you run it, all DOS disk operations will be tried as many times as you specify until you boot DOS again. (And, if you would like to make the changes permanent on a given disk, just answer the question in the program appropriately. An error message from this operation probably indicates that the disk was full, the disk was write-protected, or the DOS.SYS file was locked. In any case, try another disk.)

Finally, if you are still using DOS 2.0s (or DOS XL, most versions), just change three lines in the listing. I doubt seriously that either

Atari version is compatible with any other DOS, so you may need to experiment and/or contact the publisher of your DOS to change the retry counts. Anyway, the three lines to change are

```
JM 40 IF PEEK(1932)<>169 OR  
PEEK(1933)>10 THEN 230  
FC 100 POKE 1933,RETRY  
NM 230 REM *** APPARENTLY NO  
T DOS 2.0 ***
```

Random Ramblings

Unlike most personal computers, the Atari 8-bit machines use a hardware random number generator which is supposed to produce better random numbers than purely software schemes. Notice the words "supposed to." In practice, the hardware random number generator operates in a fairly predictable manner, and, if you were to take two values from it in very close succession ("very close" means under a dozen microseconds or so), you would find that certain values tend to follow certain other values.

Atari BASIC itself pulls a pair of hardware random numbers in very close succession. Most of the time, the results of the bias this creates are unnoticeable, especially when the range of random numbers you want is under 200 or so (for reasons too complicated to go into here). But look at what happens when you run Program 2. If BASIC's RND function produced truly random results, then you would expect the four counts that are printed to be approximately equal. They obviously aren't.

My suggested solutions: If you are a BASIC programmer having problems with Atari's randomness (and remember, most of you won't), look at Program 3. We avoid the RND function and instead read two random bytes directly from the hardware register. Because BASIC takes so long (rela-

COMPUTE! Disk Subscriptions

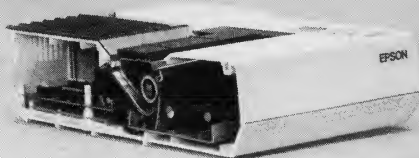
COMPUTE! Disks are available for the following computers:

- Apple II series
- Commodore 64 and 128
- Atari 400/800 /XL/XE
- IBM PC and PCjr

Each error-free disk contains all the programs from the previous three issues of *COMPUTE!*. With a disk subscription, you'll receive one disk—for the machine you specify—every three months. To subscribe, call toll free **800-247-5470** (in Iowa 800-532-1272).

Attention all FX80, FX100, JX, RX, & MX owners: You already own half of a great printer

Dealer
inquiries
welcome.



Now
Only
\$79.95

Now for \$79.95 you can own the rest. You see, today's new dot matrix printers offer a lot more.

Like an NLQ mode that makes their letters print almost as sharp as a daisy wheel. And font switching at the touch of a button in over 160 styles. But now, a Dots-Perfect

upgrade kit will make your printer work like the new models in minutes— at a fraction of their cost.

And FX, JX and MX models will print the IBM character set, too.

So, call now and use your Visa, MasterCard, or AmerEx. Don't replace your printer, upgrade it!

1-800-368-7737
In California: 1-800-831-9772

g Sample of
letter with
Dots-Perfect

Dots-Perfect™

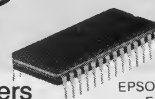
Dresselhaus

Sample of
letter without
Dots-Perfect



837 E. Alosta Ave., Glendora, CA 91740 Tel: (818) 914-5831

An upgrade kit for EPSON FX, JX, RX, & MX printers



EPSON is a trademark of
EPSON America, Inc.

tively speaking) to do this, there is plenty of time between the PEEKs, and the generator gives pretty good results. For machine language programmers who need to get two random numbers in quick succession, try a sequence like this:

LDA RNDLOC ; get first random byte
STA WSYNC ; wait for horizontal sync
LDX RNDLOC ; get second random byte

Program 1: DOS Reliability Enhancer

```

PJ 10 REM **** DOS RELIABILITY ENHANCER ****
DD 20 DIM Q$(1)
NH 30 IF PEEK(1913)<>80 AND PEEK(1913)<>87 THEN 230
KI 40 IF PEEK(1947)<>169 OR PEEK(1948)>10 THEN 230
PL 50 POKE 1913,87:REM TURN ON VERIFY MODE
CH 60 PRINT "HOW MANY TIMES SHOULD DOS RETRY"
BF 70 PRINT "A DISK OPERATION (0 TO 10) ";
IO 80 INPUT RETRY
HH 90 IF RETRY<0 OR RETRY>10 OR RETRY<>INT(RETRY) THEN RUN
FI 100 POKE 1948,RETRY
OG 110 PRINT :PRINT
PN 120 PRINT "DO YOU WANT TO MODIFY THE DOS ON"
BF 130 PRINT " THE DISK IN

```

```

DRIVE 1 (Y/N) ";
GE 140 INPUT Q$:IF Q$<>"Y" THEN STOP
FP 150 TRAP 200
AP 160 OPEN #1,8,0,"D1:DOS.SYS"
GC 170 CLOSE #1
FI 180 PRINT :PRINT "DOS MODIFIED"
HB 190 END
DB 200 PRINT "ERROR WHILE TRYING TO MODIFY DOS ON"
PH 210 PRINT " DISK IN DRIVE 1. ERROR #";PEEK(195)
GL 220 END
OB 230 REM *** APPARENTLY NOT DOS 2.5 ***
DI 240 PRINT "INCOMPATIBLE WITH THIS DOS"
BO 250 END

```

Program 2: Random Number Generator Test

```

BP 100 DIM TEST(3):FOR I=0 TO 3:TEST(I)=0:NEXT I
JE 110 FOR I=1 TO 1000
HN 120 R=INT(RND(0)*65535/128+.5)
BA 130 T=INT(R/4):R=R-T*4
PE 140 TEST(R)=TEST(R)+1
BO 150 NEXT I
CE 160 PRINT
NP 170 PRINT "VALUE", "COUNT"
AM 180 FOR I=0 TO 3
HG 190 PRINT I,TEST(I)
BK 200 NEXT I

```

Program 3: Improved Random Number Routine

```

BP 100 DIM TEST(3):FOR I=0 TO 3:TEST(I)=0:NEXT I
JE 110 FOR I=1 TO 1000
AM 120 R=PEEK(53770)*256+PEEK(53770)
BA 130 T=INT(R/4):R=R-T*4
PE 140 TEST(R)=TEST(R)+1
BO 150 NEXT I
CE 160 PRINT
NP 170 PRINT "VALUE", "COUNT"
AM 180 FOR I=0 TO 3
HG 190 PRINT I,TEST(I)
BK 200 NEXT I

```

Attention Programmers

COMPUTE! magazine is currently looking for quality articles on Commodore, Atari, Apple, and IBM computers (including the Commodore Amiga and Atari ST). If you have an interesting home application, educational program, programming utility, or game, submit it to *COMPUTE!*, P.O. Box 5406, Greensboro, NC 27403. Or write for a copy of our "Writer's Guidelines."



RUN And INIT Vectors

This month's discussion is something of a continuation of my column of a couple of months ago, where I presented a program that showed you the segments of a binary file. And that column, in turn, referred back to the April 1986 column. Both columns are required reading for a full understanding this month, but you'll learn something even if you are reading this cold.

We begin by noting that when you ask Atari DOS (version 2.0S or 2.5) to save a chunk of memory as a binary file, it asks you to supply four numbers:

START,END,INIT,RUN

And, if you've looked through enough magazine articles or user group newsletters, you've probably come across places where an author instructed you to use the save binary file option, mentioned the beginning and ending addresses, and then told you to be sure to give the proper RUN (and/or INIT) address. The START and END numbers seem obvious: They are the first and last addresses of the range of memory to be written out. But what about INIT and RUN? What can those possibly mean?

A Feature Unmatched

The ability of *any* binary file, including the ever-important AUTORUN.SYS, to have a RUN or INIT address associated with it is, in my opinion, a feature unmatched by any small system DOS, up to and including MS-DOS (IBM PC and clones) and TOS (for the ST). Only with Atari DOS's binary files and their format-compatible relatives can you tell the operating system to load part of your binary file (also called machine language file, object code, and so on—several names for the same thing), execute that part, and then continue loading more of the file. So let's see how it all works.

When DOS loads a binary file, including the AUTORUN.SYS file at power-up time, it monitors two locations. The simpler of the two is the RUN vector. Before DOS begins the load of a binary file, it puts a known value into locations 736–737 (hex \$2E0–\$2E1). When the file is completely loaded—DOS encounters the end of the file—if the contents of location 736 have been changed, then DOS assumes the new contents specify the address of the beginning of the program just loaded. DOS calls the program (via a JSR) at that address.

The second monitored location is the INIT vector, at 738–739 (hex \$2E2–\$2E3). This vector works much the same as the RUN vector, but DOS initializes and checks it for *each segment* as the segments are loaded. If the INIT vector's contents are altered, then DOS assumes the user program wants to stop the load process long enough to call a subroutine. So DOS calls (via a JSR) at the requested address, expecting that the subroutine will return so that the rest of the load can take place. This is a *very* handy feature. Most of you have probably seen it at work—for example, when a program first puts up an introductory screen (maybe just a title and a *Please wait* message) when you run (or boot), then continues to load.

Taking Full Control

The other important difference between the RUN and INIT vectors is that DOS leaves channel 1 open while the INIT routine is called. (DOS always opens and loads the binary file via this channel.) I suppose a really tricky program could close channel 1, open a different binary file, and then return to DOS. DOS would proceed to load the new file as if it were continuing the load of the original one. Most of the time, though, INIT routines should

not touch channel 1.

As noted, when you SAVE a binary file from DOS 2.x (and many of its variants), you are allowed to specify both an INIT and a RUN address. But the INIT address is sort of useless, since it is added to the end of a file; so, for example, your opening screen display won't occur until the entire file is loaded. To take full control, you must resort to assembly language (or to a compiled language, such as *Kyan Pascal* or OSS's *Action*). For those of you familiar with assembly language, I present the skeleton listing below. This listing is compatible with the *Atari Assembler Editor* cartridge or the MAC/65 assembler. You will need to make a handful of minor substitutions if you are using some other assembler.

I'm not going to explain the program in great detail—the source code is fairly well documented. A couple of important points though: Notice that there is no special command to the assembler that will force it to put in an INIT vector (or RUN vector—unless you have the AMAC assembler). Instead, we simply create a binary file segment that is only two bytes (one word) long. And this segment is loaded by DOS's loader at—where else—the appropriate vector. So the very act of loading the specified addresses modifies the contents of the vector. What could be neater?

As mentioned, this is *strictly* a program skeleton. It will do nothing as is. You must add some of your own assembly language to it to make it actually do something. So, if you thought INIT and RUN vectors were beyond you, try this skeleton and be ready to change your mind.

INIT Vector Example

; Skeleton of a program which puts
; a 'please wait' type message on

; the screen before loading the
; main code.

; * = \$3000 ; or someplace
DOINIT

; the code which follows is for
; demo purposes only! Use your
; own code...pretty display lists
; or dazzling colors or whatever

LDA #0 ; channel zero
LDA #9 ; Put Text com-
mand
STA \$342 ; command byte
LDA #MSG&255
STA \$344 ; low byte, addr
of msg
LDA #MSG/256
STA \$345 ; high byte, ditto
LDA #255 ; use a too-big
length...
STA \$348 ; since RETURN
terminates
; this call
anyway
JSR \$E456 ; call CIO
RTS ; back to DOS

MSG .BYTE 125 ; (clear screen)
.BYTE 29,29,29,29 ; (cursor down)
.BYTE 127 ; (tab once)
.BYTE "—please wait—"
.BYTE 155 ; (return...end of msg)

; now the INIT VECTOR forces DOS
; to call our DOINIT routine

* = \$2E2 ; init vector
.WORD DOWINIT ; gets pointed to us

; Your main program...
; you are on your own here!

* = \$3000 ; the same address if you
like

; I can use the same address because
; my init code can disappear when
; its job is done. This may not
; work with your code. Be careful.

DORUN

...

; then we get DOS to run our program
; by using a RUN vector.

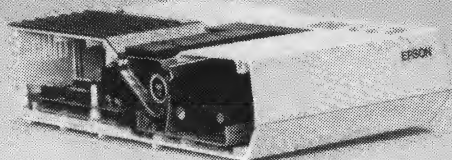
* = \$2E0 ; AMAC uses ORG,
not * =
.WORD DORUN ; AMAC uses
WORD, no dot
.END ; AMAC uses END,
no dot

©

Attention all FX80, FX100, JX, RX, & MX owners:

You already own half of a great printer

Now
Only
\$79.95



Now for \$79.95 you can own
the rest. You see, today's new dot
matrix printers offer a lot more.

Like an NLQ mode that makes
their letters print almost as sharp as
a daisy wheel. And mode switch-
ing at the touch of a button in over
160 styles. But now, a Dots-Perfect

upgrade kit will make your printer
work like the new models in min-
utes— at a fraction of their cost.

And FX, JX and MX models will
print the IBM character set, too.

So, call now and use your Visa,
MasterCard, or AmerEx. Don't
replace your printer, upgrade it!

1-800-368-7737

(Anywhere in the United States or Canada)

g Sample of
letter with
Dots-Perfect

Dots-Perfect™

Dresselhaus

8560 Vineyard Ave., Ste. 405, Rancho Cucamonga, CA 91730

Sample of
letter without
Dots-Perfect



(714) 945-5600

IBM is a registered trademark
of International Business Machines Corp.
Graftrac™ is a trademark of Epson America, Inc.
Epson is a registered trademark of Epson America, Inc.

An upgrade kit for EPSON FX, JX, RX, & MX printers



**Looking for a Widget
for your
Okidata printer and
need it now?
Call Precision!**

Precision Images normally stocks
most spare parts for your Okidata
printer, from the Okimates to the
Pacemarks including the new
Microline and Laserline series.
Anything and everything for your
Okidata printer is only a phone
call away. Precision Images is
"your direct connection to genu-
ine Okidata parts and supplies."

for Visa/MasterCard orders call:

1-800-524-8338



Precision Images, Inc.
P.O. Box 866
Mahwah, New Jersey 07430

LOTTO CIPHER™

GET THE BEST ODDS ON ANY LOTTERY
SIX NUMBER - PICK FOUR - DAILY GAME

- PRODUCES FOUR COMBINATIONS OF
NUMBERS TO CHOOSE FROM.
- ANY AMOUNT OF BALLS AND NUMBERS CAN
BE PROGRAMMED.
- PRINTS OUT PAST LOTTO NUMBERS DRAWN,
PAST COMPUTER PICKS, AND NUMBER DRAW
FREQUENCY LIST.
- RANDOM NUMBER GENERATOR INCLUDED.



\$29.95 C-64/128

Window Magic

SUPER HI-RESOLUTION DRAWING IN MULTI OR MONO COLOR

- COPY
- FILL
- LINES
- DRAW
- COLOR SQUARES
- TYPES LETTERS AND GRAPHICS
- POLYGON SHAPES-EXPAND, SHRINK AND ROTATE, THEN STAMP ANYWHERE
- ZOOM PLOT-DRAW ON AN EXPANDED WINDOW AND YOUR DRAWING AT THE
SAME TIME
- MIRROR, FLIP, AND SCROLLING WINDOWS
- ZOOM-EXPANDS A WINDOW TO DOUBLE SIZE
- SAVE AND LOAD YOUR WINDOWS ON DISK
- PRINTS ON STANDARD DOT MATRIX PRINTER
- CLONE COLOR ATTRIBUTES

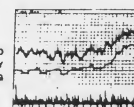
\$24.95 C-64/128

STOCK BROKER™

PROFITS GUARANTEED OR
YOUR MONEY BACK

BUYING GOOD QUALITY, VOLATILE ISSUES AND
USING THIS TRADING SYSTEM WILL HAVE YOU FULLY
INVESTED AT THE LOWEST PRICES AND CONVERTING
TO CASH AS THE STOCK NEARS ITS PEAK.

- TECHNICAL TRADING THAT WORKS
- BAR GRAPH PRINT-OUTS
- RECORD UP TO 144 STOCKS ON A DISK



\$29.95 C-64/128

ACORN OF INDIANA, INC.

2721 OHIO STREET
MICHIGAN CITY, IN 46360



219-879-2284

SHIPPING AND HANDLING, ADD \$1.50 - C.O.D.'S ACCEPTED
VISA AND MASTER CARD ORDERS ADD 4%
INDIANA RESIDENTS ADD 5% SALES TAX



RUN And INIT Vectors

This month's discussion is something of a continuation of my column of a couple of months ago, where I presented a program that showed you the segments of a binary file. And that column, in turn, referred back to the April 1986 column. Both columns are required reading for a full understanding this month, but you'll learn something even if you are reading this cold.

We begin by noting that when you ask Atari DOS (version 2.0S or 2.5) to save a chunk of memory as a binary file, it asks you to supply four numbers:

START,END,INIT,RUN

And, if you've looked through enough magazine articles or user group newsletters, you've probably come across places where an author instructed you to use the save binary file option, mentioned the beginning and ending addresses, and then told you to be sure to give the proper RUN (and/or INIT) address. The START and END numbers seem obvious: They are the first and last addresses of the range of memory to be written out. But what about INIT and RUN? What can those possibly mean?

A Feature Unmatched

The ability of *any* binary file, including the ever-important AUTORUN.SYS, to have a RUN or INIT address associated with it is, in my opinion, a feature unmatched by any small system DOS, up to and including MS-DOS (IBM PC and clones) and TOS (for the ST). Only with Atari DOS's binary files and their format-compatible relatives can you tell the operating system to load part of your binary file (also called machine language file, object code, and so on—several names for the same thing), execute that part, and then continue loading more of the file. So let's see how it all works.

When DOS loads a binary file, including the AUTORUN.SYS file at power-up time, it monitors two locations. The simpler of the two is the RUN vector. Before DOS begins the load of a binary file, it puts a known value into locations 736–737 (hex \$2E0–\$2E1). When the file is completely loaded—DOS encounters the end of the file—if the contents of location 736 have been changed, then DOS assumes the new contents specify the address of the beginning of the program just loaded. DOS calls the program (via a JSR) at that address.

The second monitored location is the INIT vector, at 738–739 (hex \$2E2–\$2E3). This vector works much the same as the RUN vector, but DOS initializes and checks it for *each segment* as the segments are loaded. If the INIT vector's contents are altered, then DOS assumes the user program wants to stop the load process long enough to call a subroutine. So DOS calls (via a JSR) at the requested address, expecting that the subroutine will return so that the rest of the load can take place. This is a *very* handy feature. Most of you have probably seen it at work—for example, when a program first puts up an introductory screen (maybe just a title and a *Please wait* message) when you run (or boot), then continues to load.

Taking Full Control

The other important difference between the RUN and INIT vectors is that DOS leaves channel 1 open while the INIT routine is called. (DOS always opens and loads the binary file via this channel.) I suppose a really tricky program could close channel 1, open a different binary file, and then return to DOS. DOS would proceed to load the new file as if it were continuing the load of the original one. Most of the time, though, INIT routines should

not touch channel 1.

As noted, when you SAVE a binary file from DOS 2.x (and many of its variants), you are allowed to specify both an INIT and a RUN address. But the INIT address is sort of useless, since it is added to the end of a file; so, for example, your opening screen display won't occur until the entire file is loaded. To take full control, you must resort to assembly language (or to a compiled language, such as *Kyan Pascal* or OSS's *Action*). For those of you familiar with assembly language, I present the skeleton listing below. This listing is compatible with the *Atari Assembler Editor* cartridge or the *MAC/65* assembler. You will need to make a handful of minor substitutions if you are using some other assembler.

I'm not going to explain the program in great detail—the source code is fairly well documented. A couple of important points though: Notice that there is no special command to the assembler that will force it to put in an INIT vector (or RUN vector—unless you have the AMAC assembler). Instead, we simply create a binary file segment that is only two bytes (one word) long. And this segment is loaded by DOS's loader at—where else—the appropriate vector. So the very act of loading the specified addresses modifies the contents of the vector. What could be neater?

As mentioned, this is *strictly* a program skeleton. It will do nothing as is. You must add some of your own assembly language to make it actually do something. So, if you thought INIT and RUN vectors were beyond you, try this skeleton and be ready to change your mind.

INIT Vector Example

; Skeleton of a program which puts
; a 'please wait' type message on

; the screen before loading the
; main code.

*= \$3000 ; or someplace
DOINIT

; the code which follows is for
; demo purposes only! Use your
; own code...pretty display lists
; or dazzling colors or whatever

```
LDX #0 ; channel zero
LDA #9 ; Put Text com-
mand
STA $342 ; command byte
LDA #MSG&255
STA $344 ; low byte, addr
of msg

LDA #MSG/256
STA $345 ; high byte, ditto
LDA #255 ; use a too-big
length...

STA $348 ; since RETURN
terminates
; this call
anyway
JSR $E456 ; call CIO
RTS ; back to DOS
```

```
MSG .BYTE 125 ; (clear screen)
.BYTE 29,29,29,29 ; (cursor down)
.BYTE 127 ; (tab once)
.BYTE "—please wait—"
.BYTE 155 ; (return...end of msg)
```

; now the INIT VECTOR forces DOS
; to call our DOINIT routine

```
*= $2E2 ; init vector
.WORD DOINIT ; gets pointed to us
```

; Your main program...
; you are on your own here!

```
*= $3000 ; the same address if you
like
```

; I can use the same address because
; my init code can disappear when
; its job is done. This may not
; work with your code. Be careful.

DORUN

...

; then we get DOS to run our program
; by using a RUN vector.

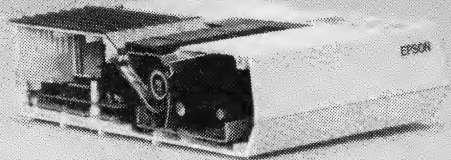
```
*= $2E0 ; AMAC uses ORG,
not *=
.WORD DORUN ; AMAC uses
WORD, no dot
.END ; AMAC uses END,
no dot
```

©

Attention all FX80, FX100, JX, RX, & MX owners

You already own half a great printer

Now
Only
\$79.95



Now for \$79.95 you can own the rest. You see, today's new dot matrix printers offer a lot more.

Like an NLQ mode that makes their letters print almost as sharp as a daisy wheel. And mode switching at the touch of a button in over 160 styles. But now, a Dots-Perfect

upgrade kit will make your printer work like the new models in minutes— at a fraction of their cost.

And FX, JX and MX models will print the IBM character set, too.

So, call now and use your Visa, MasterCard, or AmerEx. Don't replace your printer, upgrade it!

1-800-368-7737

(Anywhere in the United States or Canada)

g Sample of letter with Dots-Perfect

Dots-Perfect™

Dresselhaus

8560 Vineyard Ave., Ste. 405, Rancho Cucamonga, CA 91730

Sample of letter without Dots-Perfect



(714) 945-5600

An upgrade kit for EPSON FX, JX, RX, & MX printers

IBM is a registered trademark of International Business Machines Corp. Graphics™ is a trademark of Epson America, Inc. Epson is a registered trademark of Epson America, Inc.



**Looking for a Widget
for your
Okidata printer and
need it now?
Call Precision!**

Precision Images normally stocks most spare parts for your Okidata printer, from the Okimates to the Pacemarks including the new Microline and Laserline series. Anything and everything for your Okidata printer is only a phone call away. Precision Images is "your direct connection to *genuine* Okidata parts and supplies."

for Visa/MasterCard orders call:

1-800-524-8338



Precision Images, Inc.
P.O. Box 866
Mahwah, New Jersey 07430

LOTTO CIPHER™

GET THE BEST ODDS ON ANY LOTTERY
SIX NUMBER - PICK FOUR - DAILY GAME

- PRODUCES FOUR COMBINATIONS OF NUMBERS TO CHOOSE FROM
- ANY AMOUNT OF BALLS AND NUMBERS CAN BE PROGRAMMED
- PRINTS OUT PAST LOTTO NUMBERS DRAWN, PAST COMPUTER PICKS, AND NUMBER DRAW FREQUENCY LIST.
- RANDOM NUMBER GENERATOR INCLUDED.



\$29.95 C-64/128

Window Magic

SUPER HI-RESOLUTION DRAWING IN MULTI OR MONO COLOR

- COPY
- FILL
- LINES
- DRAW
- COLOR SQUARES
- TYPES LETTERS AND GRAPHICS
- POLYGON SHAPES-EXPAND, SHRINK AND ROTATE, THEN STAMP ANYWHERE
- ZOOM PLOT-DRAW ON AN EXPANDED WINDOW AND YOUR DRAWING AT THE SAME TIME
- MIRROR, FLIP, AND SCROLLING WINDOWS
- ZOOM-EXPANDS A WINDOW TO DOUBLE SIZE
- SAVE AND LOAD YOUR WINDOWS ON DISK
- PRINTS ON STANDARD DOT MATRIX PRINTER
- CLONE COLOR ATTRIBUTES

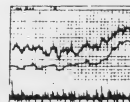
\$24.95 C-64/128

STOCK BROKER

PROFITS GUARANTEED OR
YOUR MONEY BACK

BUYING GOOD QUALITY, VOLATILE ISSUES AND USING THIS TRADING SYSTEM WILL HAVE YOU FULLY INVESTED AT THE LOWEST PRICES AND CONVERTING TO CASH AS THE STOCK NEARS ITS PEAK.

- TECHNICAL TRADING THAT WORKS
- BAR GRAPH PRINT-OUTS
- RECORD UP TO 144 STOCKS ON A DISK.



\$29.95 C-64/128

ACORN OF INDIANA, INC.

2721 OHIO STREET
MICHIGAN CITY, IN 46360



219-879-2284

SHIPPING AND HANDLING, ADD \$1.50 - C.O.D.'S ACCEPTED
VISA AND MASTER CARD ORDERS ADD 4%
INDIANA RESIDENTS ADD 5% SALES TAX



The Protection Racket

How many times have you written a program that is supposed to write to a disk file, only to have the drive make an ugly grinding noise and then have BASIC (or your program, if it is doing error trapping) tell you that the disk is write-protected? If you're like me, the answer is *very* often.

I'm usually cautious when writing my programs. I leave the write-protect tab on all my disks until I am 99 percent certain that the program will work. So during the development of a program, I tend to get lots of error number 144 messages. (Error 144 can mean anything from a too-fast disk drive to a 1050 drive's door being open, but most often it means that a disk is write-protected.)

Testing First

Wouldn't it be neat to be able to test if a disk is write-protected *before* you open a file for a write? Actually, you can. The method is a fairly obscure one; and to find it, once again I had to consult the old *Atari Technical Reference Manual*.

A short sidetrack: Those of you who don't mind searching through a couple hundred pages to find a small item buried in a lot of software (and hardware) engineering talk really should own a copy of this manual. It is now a six-year-old document, but it is still useful and accurate (aside from some of the RAM locations used by the newer XL/XE operating systems—for which you can consult *COMPUTE!'s* revised version of *Mapping the Atari*). This is a tribute to the design capabilities of the engineers from the old Atari and the astuteness of today's Atari: Never has an entire line of personal computers stayed so compatible.

Where The Secret Lies

The secret to the write-protect is

buried in the information about the disk drive's status command. (Don't confuse the drive status with a file's status, as tested by BASIC's STATUS command.) If you can recall my September 1985 column regarding SIO (Serial Input/Output) commands, it may be easier for you to understand the code which follows. I am not going into great detail, but, briefly, an SIO call must have certain information placed in page 3 (locations \$300 to \$30B, specifically) before the program jumps to the subroutine at location \$E459. (You can use my handy-dandy number converter program of a few months back to convert those hex numbers to decimal.) Information placed in page 3 includes the drive number, type of request (S, for status, in this case), the address of a buffer, and the number of bytes to transfer.

It is this last set of information that is most important to us: The drive status returns four bytes of information that we need to receive somewhere so that we can analyze it. We'll take a closer look at the accompanying listing later, but for now just notice that we dimension a buffer (BUF\$) in line 30100. The four bytes of BUF\$ will be used to receive the four bytes of drive status. We'll be using only the first byte of that status, because that is where the write-protect flag is located. Some of the bits in that byte are related to various hardware error conditions, which we need not discuss, leaving the following bits as useful to us:

Bit 3	\$08	write-protected disk
Bit 5	\$20	double density disk
Bit 7	\$80	1050 enhanced density disk

Well, well! So not only can we find out if the disk is write-protected, we can also find its density.

A Useful Subroutine

In this month's listing, then, lines

30000 and beyond are a subroutine that you can include in your own programs. To use the subroutine, simply set the variable DRIVE to a valid drive number (1 to 4, usually) and GOSUB 30000. Upon return, the variable CHECK will contain one of the following values:

less than 0	an error occurred (invalid drive number, for example)
0	disk may be written to
1	disk is write-protected

If the value returned is less than zero, then the value will be the negative of the appropriate error code. For example, if you try to check a drive that isn't turned on, the value returned should be -138, indicating that error 138—device timeout—has occurred.

Similarly, a second number is returned in a variable named DENSITY. Its meaning:

-1	density value was invalid
1	drive is single density
2	drive is double density
3	drive is enhanced density

And all of this is demonstrated by the code in lines 100 through 220. These lines are provided just to give you a test bed to try out the subroutine of lines 30000 and up. Try the program, and then incorporate the subroutine in your own programs. And never again will you or your users see that dreaded error number 144 (unless your drive speed is off—but that's another topic).

Just a couple of last comments: Notice line 30170. This demonstrates another programming trick. Worried about making sure that your RESTORE statements refer to the right DATA line numbers? For short DATA lines, why not simply combine RESTORE and DATA on the same line, as in 30170?

Also, notice the sneaky way that BUF\$ is dimensioned in line 30100. If we come to line 30100 a second time, we get an error when we try to re-DIMension BUF\$. But,

because of the TRAP, the error is effectively ignored—a useful way of making sure a variable is dimensioned only once.

Write-Protect Checker

For instructions on entering this program, please refer to "COMPUTE!'s Guide to Typing In Programs" elsewhere in this issue.

```
EA 100 REM SIMPLE PROGRAM TO
      DEMONSTRATE
BG 110 REM THE WRITE PROTECT
      CHECKER ROUTINE
HA 120 PRINT :PRINT :PRINT "
      DRIVE NUMBER ";
JQ 130 INPUT DRIVE
AJ 140 GOSUB 30100
XG 150 IF CHECK<0 THEN PRINT
      "ERROR # ";-CHECK;"
      ACCESSING DRIVE ";DRIVE:GOTO 120
CJ 160 IF CHECK>0 THEN PRINT
      "DISK IS WRITE PROTE
      CTED";
LI 170 IF CHECK=0 THEN PRINT
      "DRIVE IS READY";
PC 180 IF DENSITY=1 THEN PRI
      NT ", SINGLE DENSITY.
      "
ON 190 IF DENSITY=2 THEN PRI
      NT ", DOUBLE DENSITY.
      "
GB 200 IF DENSITY=3 THEN PRI
      NT ", ENHANCED DENSIT
      Y. "
FI 210 IF DENSITY<0 THEN PRI
      NT ", UNKNOWN DENSITY
      . "
```

```
FO 220 GOTO 100
KG 30000 REM SUBROUTINE TO C
      HECK DISK FOR WRITE
      PROTECT
NI 30010 REM
HM 30020 REM ENTER WITH DRIV
      E TO CHECK IN DRIVE
      (1 TO 8)
NK 30030 REM
JJ 30040 REM ON EXIT, CHECK
      WILL NORMALLY BE 0
LO 30050 REM IF CHECK IS 1,
      DISK IS WRITE PROTE
      CTED
HD 30060 REM IF CHECK IS <0
      THEN CHECK IS NEGAT
      IVE OF SIO ERROR CO
      DE
ON 30070 REM FOR CHECK=1 OR
      0, DENSITY IS RETUR
      NED:
JL 30080 REM DENSITY IS 1,2,
      3 FOR SINGLE, DOUBL
      E, ENHANCED DENSITY
BG 30090 REM DENSITY IS -1 I
      F UNRECOGNIZED
LK 30100 TRAP 30110:DIM BUF$
      (4)
BH 30110 POKE 768,49:POKE 76
      9,DRIVE
PI 30120 POKE 770,83:POKE 77
      1,64
CE 30130 POKE 773,INT(ADR(BU
      F$)/256)
CH 30140 POKE 772,ADR(BUF$)-
      256*PEEK(773)
JA 30150 POKE 774,2:POKE 775
      ,0
JH 30160 POKE 776,4:POKE 777
      ,0
```

```
AB 30170 RESTORE 30170:DATA
      104,76,89,228
BA 30180 FOR BYTE=1 TO 4:REA
      D CHECK
BN 30190 BUF$(BYTE)=CHR$(CHE
      CK):NEXT BYTE
NK 30200 BYTE=USR(ADR(BUF$))
BE 30210 CHECK=PEEK(771):IF
      CHECK>127 THEN CHEC
      K=-CHECK:RETURN
DJ 30220 CHECK=0:BYTE=INT(AS
      C(BUF$)/8)
GC 30230 IF BYTE/2<>INT(BYTE
      /2) THEN CHECK=1
DC 30240 BYTE=INT(BYTE/4):DE
      NSITY=-1
OH 30250 IF BYTE=0 THEN DENS
      ITY=1
OK 30260 IF BYTE=1 THEN DENS
      ITY=2
OP 30270 IF BYTE=4 THEN DENS
      ITY=3
NN 30280 RETURN
```

©

COMPUTE!'s GAZETTE
TOLL FREE
Subscription Order Line
1-800-247-5470
In IA 1-800-532-1272

Looking for Thermal Paper or Mailing Labels for your Okimates? Call Precision!

Precision Images now has avail-
able for your Okimate printers,
GENUINE Okidata thermal trans-
fer roll paper and mailing labels.
We also carry a large supply of
spare parts and supplies for all
Okidata printers. Precision Im-
ages is "your direct connection
to genuine Okidata parts and
supplies."

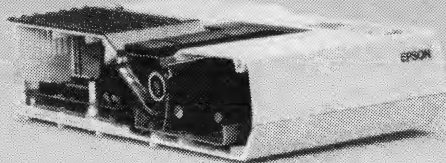
New Microline 93 Printer—\$375
for Visa/MasterCard orders call:
1-800-524-8338



Precision Images, Inc.
P.O. Box 563, Dept. C
Chester, New York 10918

Attention all FX80, FX100, JX, RX, & MX owners: You already own half of a great printer

**Now
Only
\$79.95**



Now for \$79.95 you can own
the rest. You see, today's new dot
matrix printers offer a lot more.

Like an NLQ mode that makes
their letters print almost as sharp as
a daisy wheel. And mode switch-
ing at the touch of a button in over
160 styles. But now, a Dots-Perfect

upgrade kit will make your printer
work like the new models in min-
utes— at a fraction of their cost.

And FX, JX and MX models will
print the IBM character set, too.

So, call now and use your Visa,
MasterCard, or AmerEx. Don't
replace your printer, upgrade it!

1-800-368-7737

(Anywhere in the United States or Canada)

g Sample of
letter with
Dots-Perfect

Dots-Perfect™

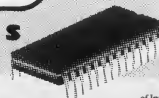
Dresselhaus

Sample of
letter without
Dots-Perfect



8560 Vineyard Ave., Ste. 405, Rancho Cucamonga, CA 91730

An upgrade kit for EPSON FX, JX, RX, & MX printers



(714) 945-5600

IBM is a registered trademark
of International Business Machines Corp.
Graphics is a trademark of Epson America, Inc.
Epson is a registered trademark of Epson America, Inc.



Three Questions

I have received a fair number of letters from 8-bit owners in recent months, and most people ask one of these three questions: "Where can I find a book that tells me . . . ?" "Do you know of any program that will . . . ?" "How do I convert my Atari BASIC program to assembly language so that I can . . . ?"

Although all your questions are slightly different, I have a few answers that will work for most of you, regardless of the ending you would like to put on any of the questions.

First, it is an unfortunate fact that many of the best books for the 8-bit Atari computers (400, 800, and XL/XE series) are no longer in print. I would like to hope that some enterprising publisher might decide to reprint a few of the best of these in limited editions, but I am not going to hold my breath until that happens. In the meantime, your best bet is to try to track down a copy *now*, while there are still a few in dealers' hands. What books am I referring to? There are so many books that would make my "nice to have" list that I can't possibly list them here, so, instead, here (reluctantly) is a limited list of what I consider my own, personal "basic necessities" library.

For all programmers:

- *COMPUTE's Mapping the Atari* and then either
- *Your Atari Computer* by Lon Poole or
- *ABC's of Atari Computers* by Dave Mentley

For BASIC neophytes:

- *Atari BASIC for Kids 8 to 80*

For assembly language neophytes:

- *Atari Roots* by Mark Andrews
- *Programming the 6502* by Rodney Zaks

For those who are *really* serious:

- Atari Technical Reference

Manuals from Atari

and, perhaps the hardest to find:

- *Atari Graphics and Arcade Game Design* by Jeff Stanton

Some of these are still pretty easy to find. Others have all but disappeared. Still, every so often I hear of dealers who have a nice stock of one or more of them. For example, you may have read in the May issue of *COMPUTE!* that B&C ComputerVisions of Santa Clara, CA, has a stock of *De Re Atari* (which just barely failed to make my essentials list). I have just learned that they also have a good stock of the Atari Technical Reference Manuals. Dealers rarely advertise that they have a certain book—by the time the ad appears, they may be sold out with no way to get more copies. So call around, ask around, check with your local bulletin board, and/or leave a message or two on some of the national time-share systems that have Atari interest areas (for example, CompuServe, Genie, and Delphi).

Ask

By now, you probably won't be surprised to find that the answer to that second question is about the same: *Ask*. About the only kind of programs you can *not* find for your 8-bit Atari are what I call "heavy-duty" programs. For example, I have yet to see a good, complete civil engineering package. Or an off-the-shelf order-entry system. The primary limitations of these small machines have always been their slow disk I/O speed and limited disk space. (Historically, there has been a more important limitation that I'll address in a future column. Ironically, I am writing this column on an 800 XL connected to a Supra 10-megabyte hard disk using ICD's *SpartaDOS*, and I find that this system now does everything I need. But a large percentage

of Atari owners have only one floppy disk drive, which is simply not enough for most business purposes.)

One amazing aspect of Atari software is the amount of usable public-domain software available. But until you join a user group—or, perhaps, buy a modem and call some BBSs or one of the national time-sharing systems—you will be cut off from this free software.

Converting BASIC To ML

The final question listed above is actually the most interesting to me: "How do I convert my Atari BASIC program into machine language?" The first and most obvious answer: *Buy a BASIC compiler*. I don't want to belabor this topic now, but you should know that Atari BASIC is an *interpreted* language. It is not fast. If you could *compile* your program into machine language, it would run much faster. (Of course, getting a better BASIC interpreter will also speed up your programs.) Remembering my advice above about finding Atari software, you *might* be able to find such a compiler. But even compiled BASIC doesn't come close to what is possible in assembly language. (Did you notice my shift from *machine* to *assembly* language? There is a technical difference between the two, but it is one we can ignore.)

Many, many articles have been written that provide you with handy machine language subroutines that you can call with Atari BASIC's *USR* function. For example, also in the May issue, Rhett Anderson presented a set of routines for doing bitwise operations via *USR* calls. The problem with most of these routines: They all tend to reside in the same hunk of Atari memory (the so-called Page 6, memory locations \$600-\$6FF, 1536-1791 decimal), so you can use only one or, perhaps, two at a time.

What happens when you need about 20 or 30 machine language subroutines? I did a whole series of articles, once upon a time, on writing self-relocatable code, machine language routines that can be loaded anyplace in memory; but it seems I was fighting a losing battle. In truth, though, it may be just as well: If you are ready to use 20 or 30 major assembly language routines, why not write the entire program in assembly language?

To do so, you need to learn two things: First, how to program in assembly language. Second, how BASIC performs its various operations. The first of these needs is answered by the books I mentioned in the first part of this article. And some of those books also go far, far beyond what Atari BASIC is capable of. But nobody seems to have written a book that shows, in a simple direct manner, how to convert the most common and useful operations of Atari BASIC into assembly language.

In particular, the topic of Atari graphics is poorly covered. There have been volumes written on display-list interrupts, player-missile graphics, custom character sets, and so on. But how does one do a simple little PLOT in assembly language? Finding the answer to that is like looking for the proverbial needle in a haystack.

When I first saw how well-designed the Atari Operating System (OS) was, I was impressed. That was more than eight years ago, and I still think it is the best OS in the world of small machines. I think you'll agree when I show you next month how little work Atari BASIC must do to perform such seemingly complex operations as GRAPHICS, PLOT, and DRAWTO. ©

COMPUTE!

TOLL FREE

Subscription

Order Line

1-800-247-5470

In IA 1-800-532-1272

Copies of articles from this publication are now available from the UMI Article Clearinghouse.

For more information about the Clearinghouse, please fill out and mail back the coupon below.

UMI Article Clearinghouse

Yes! I would like to know more about UMI Article Clearinghouse. I am interested in electronic ordering through the following system(s):

- ☐ DIALOG/Dialorder ☐ ITT Dialcom
☐ OnType ☐ OCLC ILL Subsystem

- ☐ Other (please specify) _____
☐ I am interested in sending my order by mail.
☐ Please send me your current catalog and user instructions for the system(s) I checked above.

Name _____

Title _____

Institution/Company _____

Department _____

Address _____

City _____ State _____ Zip _____

Phone (____) _____

Mail to: University Microfilms International
 300 North Zeeb Road, Box 91 Ann Arbor, MI 48106

≡≡≡CAPUTE!≡≡≡

Atari Laser Chess

A number of lines appear twice in the listing for the Atari version of this game from the June issue. When entering Program 3, simply ignore the duplicate lines 20020-21060 on page 48.

Applesoft Memory Management

The "Readers' Feedback" column from the June issue included a question about moving an Applesoft BASIC program in memory to the area above high-resolution screen page 1. There is an error in the program line provided in the answer to this question (p. 52). The final command in the line should be RUN rather than LOAD. The complete line should read as follows:

```
5 IF PEEK(104)<>64 THEN POKE 104,64:
  POKE 16384,0: PRINT CHR$(4)"RUN
  PROGRAM"
```

Font Printer For The IBM PC/PCjr

There are no corrections for any of the programs that accompany this article from the May issue (p. 79). However, the instructions for using the printing segment, Program 2, neglected to mention that disk drive names (A:, B:, and so forth) should always be entered in uppercase. Although the computer understands that A: and a: both refer to the same drive, the program does not. Also, the article states that, when using the same drive for document and font disks, the program will beep twice when it's time to change disks. Actually, only one beep is sounded. ©



Graphics: From BASIC To ML

As promised, this month I will introduce you to how Atari BASIC (and, indeed, virtually every other language available for the 8-bit Atari machines) "talks" to the Atari Operating System (OS). It is important first to note that Atari BASIC has *no* built-in graphics subroutines. All graphics support in these machines comes directly from the OS. Today, this is not much of a revelation: The Atari ST, Apple Macintosh, and Commodore Amiga machines all come complete with an OS that supports numerous sophisticated text and graphics functions. To use these features with any given language (BASIC, C, Pascal, or whatever), the language designer only needs to provide the user with a simple interface and then to let the OS do the real work.

When the Atari 8-bit machines first appeared, though, they were unique in providing this kind of interface in a family of low-priced machines. For example, the Commodore 64 has *no* graphics whatsoever built into its OS, and the Apple II (up until the IIGS) had only limited low-resolution capabilities. Generally, such machines are *not* very friendly things to write assembly language for, in marked contrast to the Atari 8-bit models. (Looking back, if we Atari loyalists have any regrets or complaints, they might be only that Atari never produced the software tools—for example, computer languages—to fully exploit the capabilities of the OS. Other companies produced those languages—such as the Pascal from Kyan Software and BASIC XL/XE from my employer, OSS—but they never achieved the success that a language from Atari could have.)

The above history lesson was not entirely an exercise in nostalgia. It leads us to a very important point: If Atari BASIC didn't have any graphics-oriented statements

built in, you could *still* perform graphics fairly easily. Let's take a look at some Atari BASIC equivalents:

10 GRAPHICS mode

is actually the same as

10 TEMP = 12: IF mode<16 THEN

TEMP=TEMP+16

11 CLOSE #6: OPEN #6,TEMP,mode,"S:"

Do you see what we did? The GRAPHICS statement is really just a special kind of OPEN. (Actually, I have left out consideration of the "+32" modes and have simplified things a bit, but for 99 percent of all programs the above will work fine.) In other words, all of the graphics modes of the Atari 8-bit computers are *built into* the OS ROMs (Read Only Memory chips).

The simplest BASIC statement to emulate is COLOR—you can leave it out altogether. We'll take a look at why when we get to PLOT in a moment. But POSITION isn't much harder (if we ignore GRAPHICS modes 8 and 24—see below).

330 POSITION xpos,ypos

can be emulated via

330 POKE 85,xpos: POKE 84,ypos

And, once POSITION has been dispensed with, PLOT becomes easy, also. (As you study these conversions, see if you can figure out why and how they need changing for GRAPHICS modes 8 and 24.)

281 PLOT xpos,ypos

becomes

281 POSITION xpos,ypos: PUT #6,mycolor

which, in turn, can be changed to

281 POKE 85,xpos: POKE 84,ypos: PUT #6,mycolor

Incidentally, *mycolor* is the color value you would otherwise have specified in the COLOR statement. And I am purposely using lowercase names for my variables

to show that the names don't matter. Choose your own as you like. Similarly, then, you can make the following substitutions.

978 LOCATE xpos,ypos,what

can be written as

978 POSITION xpos,ypos: GET #6,what

or, by extension,

978 POKE 85,xpos: POKE 84,ypos: GET #6,what

The last one, for now, is just a bit more exotic:

330 DRAWTO xpos,ypos

is actually performed as if you had used

330 POSITION xpos,ypos

331 POKE 763,mycolor: XIO 17,#6,12,0,"S:"

or, in more detail,

330 POKE 85,xpos: POKE 84,ypos

331 POKE 763,mycolor: XIO 17,#6,12,0,"S:"

Why did I go to the trouble of breaking the BASIC statements above down into equivalent forms? Because these equivalent forms are much easier to translate into assembly language, and such translation is the main point of this article. For example, POKE and PEEK are the easiest BASIC statements to translate to assembly language. The reason? Much of assembly language consists of nothing more than fancy ways to do PEEKs and POKEs. In this short set of articles, I can't hope to teach you all the addressing modes of the 6502, so let me restrict myself to the simplest form.

For a first example, to emulate the BASIC statement

POKE 85,xpos

in 6502 assembly language, we need only code

**LDA xpos
STA 85**

which can be read as "Load the A register with the contents of *xpos* and then Store the contents of the A

register into location 85." That sounds simple enough, but the thing that makes the 6502 one of the more difficult CPU's to program for is the fact that the A register can hold, *at most*, one byte of information.

You don't see why that is a problem? Well, suppose we are doing work with GRAPHICS 8 or 24, where the horizontal (x) position can range from 0 to 319. A single byte can hold only values in the range 0-255. Oops. Ah, well, the analogy with Atari BASIC isn't so bad here, because the POKE command has the same restriction. It, too, can only affect a single byte. By extension, then: How do you change more than byte when using POKES in BASIC? By using more than one POKE, right? So, in assembly language, you must use more than one STA instruction.

But if I even *hope* to be able to finish this set of articles, I will have to cut off this introduction to 6502 assembly language at this point. If you did not understand *any* of this article, I would suggest you learn more about BASIC before trying assembly language. If you are a good BASIC programmer and this left you looking for more, then I suggest that you look into some of the books I recommended in my July column.

COMPUTE! Disk Information

All the programs in this issue are available on the ready-to-load **COMPUTE! Disk**. For more information or to order an individual issue of **COMPUTE! Disk**, call toll free **800-346-6767** (in NY 212-887-8525). To order a one-year (four-disk) subscription, call toll free **800-247-5470** (in Iowa 800-532-1272).

Please specify which computer you are using.



Precision Data Products™
Complete Line of Quality Supplies for Your Computer

Precision Data Products™
POLY PACK 5 1/4" DISKETTES
(From Leading Mfr.)

- Blank Jackets
- WP Tabs
- Envelopes
- Made in USA
- 100% Error Free
- Lifetime Warranty

36¢ Each
Sold in Lots of 100 Only

SALE!

SONY POLY PACK 3.5" DISKETTES
SS 135TPI \$1.10 Each
DS 135TPI \$1.23 Each
Sold in lots of 50 only.

PERSONAL COMPUTER TOOL KIT

vinyl zipper case.
Includes 10 tools and spare parts container.

Just \$19.95 plus \$2.00 S&H

Discounts on Larger Quantities
Min. Order \$25.00. S&H: Continental USA \$4.00/100 or fewer disks. Reduced shipping charge on larger quantities. Foreign orders, APO/FPO, please call. MI residents add 4% tax. Prices subject to change without notice. Hours: 8:30 AM - 7:00 PM ET.

Precision Data Products™
P.O. Box 8367, Grand Rapids, MI 49518
Customer Service & Information:
(616) 452-3457
Toll-Free Order Lines: FAX (616) 452-4914
MI 1-800-632-2468/Outside MI 1-800-258-0028

WP-KEY WP-KEY WP-KEY

WP-KEY

A NEW FORM OF WRITE PROTECTION

WP-KEY slides into the disk at the top corner directly above the write-protect notch and slips down to cover the notch from the inside.

WP-KEY slides out just as easily to allow the disk to be written to.

WP-KEY is good for the life of your disk.

WP-KEY is light in color allowing it to be viewed when the disk is installed in the drive just by glancing at the drive door.

WP-KEY sells for \$1.39 per pkg of ten (10). Please include \$.50 for shipping and handling. Sorry, no C.O.D. or credit card orders.

ORDER TODAY!

WRITE PROTECT



STORAGE

NEAR FUTURE COMPUTER
P.O. Box 1726
Walla Walla, WA 99362
(509) 525-3288

Did you know some of the very BEST software is available as SHAREWARE?

Shareware is a revolutionary concept of bringing top quality software to users at a fraction of the commercial prices. These programs request that the satisfied user register with the author to receive program updates, technical assistance and additional documentation.

Sample some of the many programs available through the PC-SIG library at only **\$6 per disk**

WORD PROCESSING

- ☐ #78, #627 PC-WRITE: A powerful and popular word processing program; works well with a mouse. Spelling checker is an additional feature \$12.
- ☐ #528 New York Word: This program offers mail merge, test buffering, split screens, and a host of special features \$6
- ☐ #455, #681, #682 PC TYPE+: This processor is from Jim Button's library. Easy to use and combines easily with PC-FILE+ for mail merge \$18

DATABASE MANAGEMENT

- ☐ #5, #730 PC FILE+: A powerful database management program that is easy to use. Great for creating labels. Works well with PC TYPE+ to generate forms and letters \$12
- ☐ #287, #288 FILE EXPRESS: An information management program. Small and medium sized databases are easily manipulated by using menu driven commands \$12

UTILITIES

- ☐ #478 HARD DISK UTILITIES: A selection of the best utility programs have been selected from over 25 disks in the library and compiled onto one disk for your ease of use \$6
- ☐ #517 IMAGEPRINT: This program allows your dot matrix printer to generate documents that are in Near Letter Quality format \$6

COMMUNICATIONS

- ☐ #499 PROCOMM: A professional communications program \$6

LANGUAGES/EDUCATIONAL TOOLS

- ☐ #612 FOREIGN LANGUAGES: A self-learning educational tool; provides instruction in Hebrew, German, Spanish, French, and Italian. Multiple choice format keeps track of errors for retesting purposes. Beginning and intermediate levels \$6

SALES MANAGEMENT

- ☐ #687, #688, #689 PROSPECT: A useful sales management program that allows you to schedule appointments. Searches and retrieves names, activities, or messages with ease and speed \$18

GAMES

- ☐ #390 FLIGHTMARE: An excellent graphics game. Highly recommended to test your skills \$6
- ☐ #457 GREATEST ARCADE GAMES: This includes Striker and Space Wars, and many more A must for your game library \$6

SPREADSHEETS

- ☐ #199 PC CALC: This program is easier to use than Lotus; very powerful and effective \$6
- ☐ #696 QUEBECALC: This spreadsheet is three dimensional, with an X, Y, and Z axis. Commands are similar to Lotus \$6

- ☐ PC-SIG LIBRARY ON CD ROM \$295.00
- ☐ NEW 1987 PC-SIG DIRECTORY \$12.95
- ☐ 1 YEAR PC-SIG MEMBERSHIP \$20.00
(\$36 Foreign) (Includes directory, bimonthly magazine and more.)

SPECIAL OFFER

Any 5 Disks plus 1-Year Membership
Only \$39 (Include \$4 shipping & handling)

Most programs have documentation on disk and request a donation from satisfied users. Please add \$4 postage and handling per order (\$10 foreign) — California residents add state sales tax.

414

Total Enclosed \$_____ by ☐ Check ☐ VISA ☐ MC

Card No. _____

Exp. date. _____ Signature _____

Name _____

Address _____

City _____ State _____ Zip _____



To order, call: 800-245-6717

In CA: 800-222-2996

For technical questions or local orders: (408) 730-9291

1030D East Duane Avenue Sunnyvale, CA 94086



Machine Language Graphics

Last month we looked at how Atari BASIC translates its own graphics-oriented statements into simpler pieces for its calls to the Atari's operating system (the OS). Or, more correctly, we showed how *you* could do such an expansion. When Atari BASIC executes a statement in your program, it actually interprets it as a request to do a series of machine language operations—the equivalents of the simplified pieces we discussed last month.

The only example we've taken a close look at so far is POKE. I showed you that

POKE 85,xpos

may be accomplished by the machine language instructions

**LDA xpos
STA 85**

(Remember, I'm using variable names with lowercase letters on purpose, to remind you that the names are arbitrary. Please pick your own.)

Again, if you go back to last month's column, you'll find that the only BASIC statements I used to simulate the graphics commands of BASIC were OPEN, CLOSE, PUT, GET, and XIO. You may also have noted that each of these statements was associated with a channel number (specifically, channel 6, because that's where BASIC does all its graphics operations). You won't be too surprised, then, when I tell you that each of these five is actually a fundamental OS operation. Specifically, each involves a direct call to Atari's Central Input/Output (CIO) processor. You may, however, be a little startled when I tell you these five calls represent all but one of the fundamental OS operations. (The missing one is represented by BASIC's STATUS statement, which is generally used only for modem operations because of a flaw in BASIC's implementation

of the OS call.)

The point of all this is both simple and important: If you master these five OS calls from machine language, you can use virtually any input/output (I/O) operations you might need or want. For example, you can read records from a disk file using only three of these operations (OPEN, GET, and CLOSE). True, there are some variations on GET and PUT that are useful with lines of text or with large files, but the concepts are all the same. So, without further delay, let's translate the five BASIC I/O statements into five machine language routines.

All I/O on the Atari is controlled through eight Input/Output Control Blocks (IOCBs), one for each *channel* or *file number*. Each IOCB is 16 bytes long and is located adjacent to another, beginning at addresses 832, 848, 864, and so on. (In hexadecimal, the sequence is \$340, \$350, \$360, and so on.) The channels are numbered 0-7 in BASIC, but in machine language,

we use the offset from the start of the first IOCB as the IOCB number. Under this system, the first block is still IOCB 0, but the fourth, known as channel 3 in BASIC, is designated as IOCB number 48 (\$30). The reason for this is because it begins at memory location 880 (\$370), which is 48 bytes beyond the start of IOCBs at location 832.

Graphics I/O

To perform any I/O operation, you put information into certain places in the IOCB of your choice. Then you put the IOCB number into the processor's X register and call the CIO routine at address \$E456 in ROM. (I'm not going to put in the decimal equivalents from now on. You really should learn to use hexadecimal—it's much more logical for machine language.) The only magic, then, is in learning just *what* to put into the IOCBs.

Each IOCB consists of 16 bytes, as shown in Table 1.

All of these labels and bytes

Table 1

Label Name	Size in bytes	Offset in IOCB	Mnemonic Description
ICHID	1	0	Identifier
ICDNO	1	1	Device number
ICCOM	1	2	Command
ICSTA	1	3	Status
ICBA	2	4	Buffer address
ICPT	2	6	Put vector
ICBL	2	8	Buffer length
ICAX1	1	10	Auxiliary byte 1
ICAX2	1	11	Auxiliary byte 2
ICAX3	1	12	Auxiliary byte 3
ICAX4	1	13	Auxiliary byte 4
ICAX5	1	14	Auxiliary byte 5
ICAX6	1	15	Auxiliary byte 6

Table 2

Command	ICCOM	ICBA	ICBL	ICAX1	ICAX2
OPEN	3	device	X	type	mode
CLOSE	12	X	X	X	X
GET	7	X	\$0000	===	===
PUT	11	X	\$0000	===	===
XIO	xio	device	X	===	===

have uses (I refer you to *Mapping the Atari*, or *Atari Roots* for more details), but for our purposes, we need to learn about only a few of them. Again, I have prepared a chart (Table 2) to summarize which labels are meaningful for which graphics-related commands. (Remember, see last month's column for examples of the BASIC commands we are using.) If a labeled location has a number assigned to it, then use that number with the operation. Descriptions in italics (*device*, for example) will be explained in the text that follows. An X means that the value in the corresponding location has no effect for the operation, and === means that the contents of the corresponding location should not be disturbed. For our purposes, these two symbols are equivalent: We won't change the contents of these locations.

CLOSE is the simplest of the routines. To do a CLOSE, you simply place the command number in the appropriate location, load the X register properly, and call CIO. The complete routine looks like this:

```
LDX #$60      ; using channel 6—graphics
LDA #12       ; CLOSE command
STA ICCOM,X   ; put command in place
JSR $E456     ; call CIO
```

Don't understand all that? Don't worry. A few sessions with an assembler and a good tutorial will help you get started.

For OPEN and XIO, the buffer address (ICBA) field should contain the address in memory of the beginning of a string, and that string should have the name of the device (and/or file) that you wish to work with. For graphics, the device name is always S:. The command value (ICCOM) is always 3, for OPEN. For XIO, you use the same number you would in BASIC. (For example, 17 for DRAWTO, as we saw last month.)

For OPEN, the first two auxiliary bytes (ICAX1 and ICAX2) correspond to the two auxiliary values in the BASIC version of the statement. Although ICAX2 is usually given a zero value, when opening a graphics screen, it gets the number of the appropriate graphics mode instead. Usually no command, except OPEN, should touch the auxiliary two bytes. (Atari BASIC actually errs in making them part of

the normal XIO commands, and that's why we had to stick in a value of 12 in our DRAWTO equivalent last month. The exceptions that prove the rule are various modem command XIOs, used with the R: device.)

Finally, for GET and PUT, as we will use them for graphics, you need only put a value of zero in both bytes of the buffer length (ICBL), put the appropriate command value (7 or 11) in its field (ICCOM), set up the X register, and use the A register to transfer the byte. That is, if you want to PUT a byte to the screen—which, as I hope you remember from last month, is how you implement PLOT—put the byte (for example, the color value) in the A register just before calling CIO. If you want to GET a byte from the screen to simulate the LOCATE command, do all of the above and the byte will be in the A register after your call to CIO.

Too complicated? Cheer up. This is the worst of it. Next month we'll put together some bona fide examples to try out. Next month will also be the last part of this series on converting BASIC graphics commands to machine language. I intended all of this to be an introduction (or refresher, for you old-timers) to machine language. If you want to take this topic further, you really *must* get an assembler and a couple of books. Good luck. ©

All the programs in this issue are available on the ready-to-load **COMPUTE!** Disk. To order a one-year (four-disk) subscription, call toll free
1-800-727-6937.
 Please specify which computer you are using.

Save Your Copies of COMPUTE!

Protect your back issues of *COMPUTE!* in durable binders or library cases. Each binder or case is custom-made in flag-blue binding with embossed white lettering. Each holds a year of *COMPUTE!*. Order several and keep your issues of *COMPUTE!* neatly organized for quick reference. (These binders make great gifts, too!)



Binders

\$9.95 each;
 3 for \$27.95;
 6 for \$52.95

Cases:

\$7.95 each;
 3 for \$21.95;
 6 for \$39.95

(Please add \$2.50 per unit for orders outside the U.S.)

Send in your prepaid order with the attached coupon

Mail to: Jesse Jones Industries
 P.O. Box 5120
 Dept. Code COTE
 Philadelphia, PA 19141

Please send me _____ *COMPUTE!* ☐ cases ☐ binders.
 Enclosed is my check or money order for \$ _____ (U.S. funds only.)

Name _____
 Address _____
 City _____
 State _____ Zip _____

Satisfaction guaranteed or money refunded.
 Please allow 4-6 weeks for delivery.



Atari's Newest Drive

After three months of pretty heavy stuff, it's time for a slightly different tack. And since my time has recently been monopolized by a project near and dear to all eight-bit Atari owners, I've decided to share some "secrets" with you. We're going to take a very close look at the new XF551 drive from Atari.

The XF551 is a sleek drive, lower and wider than a 1050, and in a style and color that matches the XE computers. Quite simply, it looks good. As you read about the internals of the drive, I hope I can convince you that Atari has really done something *right*.

The XF551 started out as the XF351—the 3 designated a 3½-inch drive. Some people are disappointed that Atari changed over to a 5¼-inch drive, but I view it as a very positive step. Current users can upgrade to this drive, yet still keep and use all their old disks. Software manufacturers don't have to produce two different versions of their software, and there are other points of compatibility.

For starters, the drive is compatible with disks created by virtually all Atari-compatible drives—in single, enhanced, and double density. Not only that, several of the different DOS systems I've tried have also worked flawlessly. And I know Atari has tested the drive with many, many pieces of commercial software with many different protection schemes. Summary: The drive works, and works well.

At a suggested price of under \$200, the very fact that a true double-density drive is now available from Atari would be welcome news. But the drive is also double-sided. That means that each disk can hold up to 360K—nearly three times the capacity of a 1050 and four times that of an 810.

As I write this article, Atari does not have a DOS that will sup-

port this extra capacity. However, the reason this drive has monopolized my time recently is simple—I have been writing a new DOS for Atari. ADOS (as it will be known) is full-featured, with subdirectories, random access files, a combination menu/command structure, and much more. However, it is not releasable as I write this, so back to the drive.

Inside The Drive

As you may remember, I discussed SIO (Serial Input Output) as it applies to disk drives, in the September 1985 issue. I noted that the four basic SIO commands are R, W, P, and S, for Read, Write, Put, and Status, respectively. Besides these, the Atari 810 and 1050 only understand format commands.

Then, in the next issue, I explained the concept of a device configuration table, as implemented by all the makers of true double-density drives. Well, we can add Atari Corporation to that list: The XF551 supports the Percom standard configuration table. That means you can tell the drive that it's an 810, a 1050, a double-density drive, or (best of all) a double-sided double-density drive. Or, perhaps just as important, the drive can tell you what kind of disk it holds. For these capabilities, we add N and O (which I think of as iN and Out) commands on the serial bus.

But there's even more. If you send it a Read or Write or Put command with the upper bit set (the inverse video bit, in screen terms), then the XF551 transfers data in high-speed mode. To take advantage of this, you need a compatible DOS, but ADOS is nearly ready and I'm sure others will be modified to support high-speed transfers.

Last, but not least, the XF551 adds a special format command (hex \$A1, an inverse-video exclamation

point) that tells the drive to use a special high-speed interleave that enhances the high-speed read and write commands even more. (But note that ordinary reads and writes are even slower than usual on disks formatted in this special way, just as they are on Sparta DOS ultraskew disks used in non-US Doubler drives. I should warn you that each of these drives seems to use a slightly different high-speed scheme.)

So the drive gets my nod of approval from a software standpoint. But what about the hardware? Will the drive stand up to physical abuse, overheating, and the like? Truthfully, I have not had even the prototype long enough to make a definitive statement on this point. But I *have* had the cover off the drive, and I have looked at its construction. It looks great. The inside is as well built as the outside.

In fact, Atari has never produced a more solid piece of equipment. The drive frame is heavy-duty cast aluminum, the mechanical parts are finely polished and aligned, and the controller board appears to be adequately ventilated. Only one point of caution: Double-headed drives are more sensitive to shock and misalignment than their single-headed cousins. Treat the drive with care. Always use its cardboard protector when you move it. Make sure it has adequate ventilation. In other words, use common sense.

If this column sounds like an advertisement for the XF551, I won't apologize—I'm not getting a penny in royalties on the drive or ADOS. This glowing report is for one reason and one reason only: I just *had* to tell you that Atari has not abandoned the eight-bit market. And they've proven that fact in grand style. ©



Crosswords And Home Computers

I recently attended a meeting of the ESCAPE user's group (of Santa Cruz, California) and had the pleasure of participating in a lively discussion of programming in general, and program design in particular. I must give John Pilge credit for starting the session off with an example program that I'm still thinking so hard about. I would like to share his tough nut with all of you.

Consider a simple acrostic square, such as this:

T	O	N
A	R	E
B	E	T

There are six interlocking words in the square, three vertically and three horizontally. The problem: Given a list of three-letter words, can you write a computer program that will create an acrostic square? Better yet, given a list of five-letter words, can your program use ten of those words to create a larger acrostic square?

John did, indeed, produce a program to perform that task, but when he gave it a list of several thousand five-letter words to work with, and then performed some speed measurements, he estimated that his Atari 800 computer would take 67 years to finish! What can be done to improve that time? The most obvious answer would seem to be to move up to a faster machine. Why not? An Atari ST might even be able to do the job in just a handful of years. Hmmm...not good enough yet? I didn't think so either.

So now we are into the meat of this month's column: How can we write programs to get the best possible performance out of our little beasts?

Speed Demons

For starters, John *did* write his program in BASIC. Now, as convenient as BASIC is, it is certainly not a

language to use when you need speed, so let's consider using another language. Typically, compiled languages will run programs from 10 to 200 times faster than interpreted BASIC (depending on what compiler you use and what kind of program you are testing).

If you are ready to resort to assembly language, you can improve those numbers by an additional factor ranging from 2 to 20. Still, that means that even at *best*, a change of languages will get us an improvement of no more than, say, 400 to 1 over BASIC. So, 67 years becomes about two months. Sigh. I'm not sure I could do without my machine for that long.

Besides changing languages or computers, there are two ways to attack the problem of a too-slow program. The first is to examine your code carefully, looking for the little things that slow down the system. For example, most of us have learned that with Atari BASIC (and indeed, with most BASICs), you can improve performance markedly if you put FOR...NEXT loops at the beginning of your program. Yet sometimes, even that is not enough. As I mentioned once upon a time in an article about card shuffling, the only real solution might be to find another *method* to solve the problem.

Try the listing accompanying this article, as is. Note how quickly the program finds the acrostic squares. Then, run it twice more, removing lines 1430 and 1450, in turn, to see the effect of increasing the word list even a little bit. Now, imagine the effect of having thousands of words. Worse, imagine thousands of five-letter words. Scare you a little, doesn't it?

In the same vein, consider crossword puzzles. If five-square acrostics are a tough nut for computers, imagine how long it would take your Atari computer to gener-

12 × 12 crossword. Even today, there is no *real* crossword-generating program for any personal computer. (Yes, I am aware of *Crossword Magic*, but that program only *aids* crossword makers. It doesn't even come close to being able to generate puzzles by itself.) Yet, there are humans who can produce original crossword puzzles in a matter of hours or even minutes.

For you nonprogrammers, I hope I've shown you that computers can't do everything as well as humans. There is a not-too-hidden message here as well: Program design is very important. Yes, careful implementation is important (no one wants a buggy program, of course), but sometimes a good design can make the *real* difference, hopefully producing a program that can finish its task before you nod off with boredom.

For you programmers, here is a challenge: Can you come up with a better method? I would hope so. My version is fairly simplistic (probably much more so than John's program, which I have not seen) and not too hard to follow. I do have a sneaking hunch, though, that you won't improve the program too much if you stick to BASIC—not because BASIC itself is slow (although that doesn't help)—but because BASIC is so weak when it comes to data types and structures.

Foreign Languages

And where is this leading us? Into one of my favorite topics: computer languages. More specifically, I would like to explore the strengths and weaknesses of the various languages available for Atari computers (both eight-bit and ST). Beyond that, I would like to discuss some of the more fundamental programming topics. In particular, in next month's column we will begin looking at the advantages of struc-

tured data (and no, that's *not* the same as structured programs).

In the meantime, I'm going to be giving the acrostic squares problem some thought. Certainly, if you get any brilliant ideas for solving the acrostic squares problems, please let me know. You can write to me at P.O. Box 710352, San Jose, CA, 95171-710352. Or, you can contact me in one of the Atari forums on CompuServe. (I am active in the Atari eight-bit forum, especially since they introduced the Kyan/OSS/ICD special topics areas. My PPN is 73177,2714.) And please, if your solution is a lengthy one, consider sending a disk or uploading the program.

```
AD 10 REM COPYRIGHT 1987 COM
PUTE! PUBLICATIONS, IN
C. ALL RIGHTS RESERVE
D.
IP 20 PRINT "{CLEAR}COPYRIGH
T 1987":PRINT "COMPUTE
! PUBLICATIONS, INC.":
PRINT "ALL RIGHTS RESE
RVED."
BP 30 FOR TT=1 TO 1200:NEXT
TT:PRINT "{CLEAR}"
FJ 1000 REM THE HORIZONTAL W
ORDS:
```

```
FK 1001 DIM H1$(3),H2$(3),H3
$(3)
KK 1010 REM THE VERTICAL WOR
DS:
IF 1011 DIM V1$(3),V2$(3),V3
$(3)
MC 1020 REM A TEMPORARY AND
MASTER WORD LIST
IA 1021 DIM T$(3),W$(3*100)
LP 1030 REM INITIALIZE THE M
ASTER WORD LIST
MB 1040 FOR I=1 TO 1000000 S
TEP 3
FK 1050 READ T$
HB 1060 IF T$<>"*" THEN W$(I
)=T$:NEXT I
EN 1070 REM NOW BEGIN THE RE
AL WORK
OL 1080 WCNT=I-3
OG 1090 FOR H1=1 TO WCNT STE
P 3
OD 1100 H1$=W$(H1,H1+2)
OA 1110 FOR H2=1 TO WCNT STE
P 3
HL 1120 IF H2=H1 THEN 1350
OJ 1130 H2$=W$(H2,H2+2)
OE 1140 FOR H3=1 TO WCNT STE
P 3
FA 1150 IF H3=H2 OR H3=H1 TH
EN 1330
OP 1160 H3$=W$(H3,H3+2)
FH 1170 V1$=H1$(1):V1$(2)=H2
$(1):V1$(3)=H3$(1)
PE 1180 FOR V1=1 TO WCNT STE
P 3
BA 1190 IF V1$<>W$(V1,V1+2)
THEN NEXT V1:GOTO 13
20
FM 1200 V2$=H1$(2):V2$(2)=H2
$(2):V2$(3)=H3$(2)
```

```
OP 1210 FOR V2=1 TO WCNT STE
P 3
AO 1220 IF V2$<>W$(V2,V2+2)
THEN NEXT V2:GOTO 13
20
BF 1230 V3$=H1$(3):V3$(2)=H2
$(3):V3$(3)=H3$(3)
PD 1240 FOR V3=1 TO WCNT STE
P 3
BF 1250 IF V3$<>W$(V3,V3+2)
THEN NEXT V3:GOTO 13
20
BJ 1260 PRINT "FOUND ONE!"
FH 1270 PRINT
CB 1280 PRINT ,H1$
CD 1290 PRINT ,H2$
BM 1300 PRINT ,H3$
FC 1310 PRINT
NN 1320 REM (TO HERE IF VN$
NOT IN LIST)
AB 1330 REM (END OF 'IF H3=H
2 ...')
IC 1340 NEXT H3
AB 1350 REM (END OF 'IF H2=H
1 ...')
ID 1360 NEXT H2
ID 1370 NEXT H1
BC 1380 STOP
LB 1390 REM
DK 1400 REM THE WORDS!
KK 1410 REM
AF 1420 DATA ARE,BET,NET,ORE
,TAB,TON
AM 1430 DATA *
DJ 1440 DATA TOP,PET,TAP,POT
,TAN,PEN
AO 1450 DATA *
ON 1460 DATA LAP,LOP,CAP,COT
,CAT,CAN
BA 1470 DATA * ©
```

MICRO WORLD ELECTRONIX



NEW 500!

512K Computer with 1 Disk Drive,
1080 Color Monitor. Includes
Software

AMIGA 500 CALL
1MEG RAM EXP CALL
EXTERNAL DRIVE CALL

AMIGA 2000 NOW SHIPPING!!

Peripherals now in stock:

A2088D Bridgecard
A2090 HD Controller
A2092 20MB HD W/Cont
A2052 2 MB Board
A2010 3.5" Disk Drive
A2002-23 Monitor
A1010 External Floppy

EPSON

LX800 \$175.95
FX86E \$CALL
FX286E \$CALL
EX800 \$CALL
EX1000 \$504.00
LQ800 \$LOW
LQ850 \$495.00
LQ1000 \$505.00
LQ1050 \$669.00
LQ2500 \$SAVE
GQ3500 laser \$CALL

PANASONIC

1080I MODEL II \$159.95
1091I MODEL II \$CALL
1092I \$295.00
1592 \$379.00
1595 \$CALL
3131 \$249.00
3151 \$CALL
4450 LASER \$LOW
1524 24 PIN \$SAVE\$

FREE

**DECEMBER SPECIAL!!! BUY ANY TWO ELECTRONIC ARTS TITLES
AND RECEIVE A FREE TEE SHIRT OR CAP!!!**

FREE

Electronic Arts

Bard's Tale \$31.95
King's Quest I, II, III \$31.95
Space Quest \$31.95
Leisure Suit Larry \$31.95
Marble Madness \$31.95
Deluxe Paint II \$79.95
Deluxe Print \$69.95
Deluxe Music \$64.95
Deluxe Video \$79.95
Ferrari Formula I \$31.95
Earl Weaver Baseball \$31.95
Gridiron! \$42.95
Chessmaster 2000 \$29.95
MathTalk \$31.95
Empire \$31.95

Microillusions

Faery Tale Adventure \$31.95

AMIGA SOFTWARE

Mindscape

Dejavu \$31.95
Brattacus \$31.95
Defender of Crown \$31.95
SDI \$31.95
Sinbad \$31.95
King of Chicago \$31.95
Uninvited \$24.95
SAT Prep. \$39.95

Microdeal

GoldRunner \$26.95

Gamestar

Baseball \$25.95
Football \$25.95

Graphics

Sculpt 3D \$64.95
Aegis Animator \$84.95
Aegis Draw Plus \$159.00

Business Software

VIP Professional \$159.95
Superbase \$89.95
Word Perfect \$199.95
Rags to Riches Acc. \$CALL
Phasar \$CALL

64 & 128 Software

Paperclip \$39.95
Superbase \$59.95
Bard's Tale \$26.95
Bard's Tale II \$19.95
Dan Dare \$19.95
Delta Patrol \$21.95
Marble Madness \$21.95
Legacy of Ancients \$21.95
Patton VS Rommel \$21.95
Pegasus \$14.95
Pegasus ADV. Battle \$22.95
Skate or Die \$22.95
Strike Fleet \$22.95

C64 & 128 ACCESSORIES

MW-350 Parallel Printer Interface

2K BUFFER \$49.95

10K BUFFER \$59.95

C64 Power Supplies

Repairable \$39.95

Non- Repairable \$29.95

MW-256

**256K Parallel Printer Buffer
\$129.00**

MW-232 C64/128 RS232 Serial Interface \$45.00
MW-401 40/80 Column Cable for C128 \$29.95
MW-611 Universal I/O /A to D converter \$225.00



SALES: 1-800-288-8088 TECH SUPPORT: (303) 988-5907
Manufacturer's Warranty Honored. All Prices Subject to Change Without Notice.



Beyond BASIC

In my last column, I promised that this month would mark the beginning of a discussion of computer languages. In particular, I want to take a look at the strengths and weaknesses of various languages. So this month, I'll open the mini-series by looking at data types.

If BASIC is your only programming language, then you probably have not run across this topic before. Yet, more than likely, you have already used various data types. In BASIC, the two underlying data types are numbers and strings. Typically, you might write program lines such as these, where the first line demonstrates numeric data types and the second shows strings:

```
TOTAL = 3.7 * SUBTOTAL
FILES$ = "D3:TEST.DAT"
```

Because BASIC has only these two types, the language has a very simple scheme for distinguishing them: String variables have a dollar sign on the end of their names, and string literals have quotation marks around their contents. Other data items are assumed to be numeric—simple and clean. Yet even in BASIC there are actually several *implied* data types that are not specially declared.

For example, consider the address that you PEEK or POKE to. It must be a number between 0 and 65535. The actual value at that address must be a number between 0 and 255. File numbers must be between 0 and 7. The list could go on. You object? You say these are all simply restricted ranges of the basic numeric data type? In BASIC, that is true. But in other languages. . .

Just My Type

Consider the following fragment of a Pascal program. In Pascal, the keyword TYPE means that the following declarations are naming various *kinds* of data, not reserving actual data space. The keyword VAR

means that further declarations do indeed reserve space for variables.

```
TYPE
  Mem_Address = 0..65535;
  Mem_Data = 0..255;
  Channel = 0..7;
  Open_Mode = ( Rd,Wr,Up )
  Cust_Rec = RECORD
    Name : String[30];
    Addr : String[30];
    City : String[15];
    State : String[2];
    Zip : 0..99999;
    Credit : ( OK,Avg,Bad );
  END;
```

```
VAR
  Peeker : Mem_Address;
  Peeked : Mem_Data;
  Customer : Cust_Rec;
  Mail_List : ARRAY [1..100]
    OF Cust_Rec;
```

Do you see what we have done? Thanks to Pascal's very rich data-typing capability, we are able to explicitly say what *kinds* of things a given variable is expected to handle. Take a close look at the variable *Peeked*. Its declaration says that it is a memory data type. Most Pascals will not even let you *try* to do a statement such as this:

```
Peeked = 3.7 * Total;
```

You are trying to assign a number that probably has a fractional part to a variable that can only have integer values from 0 to 255. Pascal knows you are being naughty, and the compiler burps real quick! And, although the following statement might get through the compiler, it will probably get you a range error when you run the program (if the original value of *Peeked* is 2 or more):

```
Peeked = 243 * Peeked;
```

Wow! Safety first, right? Well, yes. But it is more than that. Code written with strong data typing is more likely to run correctly (I have had several Pascal programs that worked the first time, once they had successfully compiled). Most importantly, in a commercial environment, such code is *maintain-*

able—a programmer can look at the code months or even years later and figure out what it is doing.

So, without even really trying, I have shown you one reason to consider learning languages other than BASIC. And I did not mean to imply that Pascal is the only language that has advantages here. Although C is generally more forgiving (another way of saying you can shoot yourself in the foot more easily) than Pascal, you can build quite readable and properly declared data types and structures with it. And, in fact, the newer versions of C—ones which follow the proposed ANSI standard—offer an option of choosing all the close checking of Pascal.

Setting The Record Straight

Go back and look at those Pascal data type declarations again. In particular, look at the *Cust_Rec* type and the *Customer* and *Mail_List* variables. Just as Pascal allows more restrictive variables than BASIC, so does it allow more complex variables. Consider these legal Pascal statements (given the above declarations):

```
Customer.Name := 'Jones';
Customer.Zip := 77344;
Mail_List[7] := Customer;
IF Customer.Credit = Bad
  THEN Write('No Credit!');
```

Those first two lines might find their way into a BASIC program looking something like this:

```
CUST$(1,30) = "JONES"
CUST$(78,82) = STR$(77344)
```

Which is more readable? If you decided to change from 5-digit to 9-digit zip codes, which program do you think would be easier to modify? No contest, right? And how would you begin to do something as simply as those third and fourth statements in BASIC? ©



Adding Power To BASIC

Last month we took a look at data types and how they're used successfully and profitably in computer languages such as Pascal. There are real and discernible advantages to using a language that handles structured data, and I hope I convinced you of that. Of course, most of those languages offer other significant reasons to use them, such as faster execution speed. Still, none of them do one thing as well as good old BASIC does. The interpretive environment of BASIC makes program development exceptionally easy.

When I travel to user-group meetings and show off one or the other of the OSS advanced BASICs, I inevitably write a program that I make up on the spot, in the meeting. And I usually manage to convert a few nonprogrammers into at least thinking about taking up BASIC as a hobby. I'm not sure I could do that with most compiler environments. So, like it or not, I do understand why most people want to learn to program in BASIC first. (And did you notice that I didn't even mention the usual reason? BASIC comes with the machine, so you don't have to pay extra to use it.)

So, given that most of you, my readers, will program in BASIC, the least I can do is show you some techniques to make such programming easier. To me, that implies showing you how to use techniques from other languages in BASIC. In turn, that means learning some tricks that will make BASIC more powerful.

Sorting Things Out

Type in and try Program 1. When you run it, give it any numbers you like, including perhaps several occurrences of the same value. When you finally enter a 0 value, the program will print out the list of your numbers in sorted order. Congratulations—you've just used a tech-

nique known as an *insertion sort*.

The name makes sense, doesn't it? As each new number is entered (line 25), we find where it belongs (that is, after which current number; lines 35 and 40), and then "insert" it into the appropriate spot in the list of numbers (lines 45 through 60). In some situations, this is a pretty good sorting method. For example, when you have to wait several seconds between user input, what's a quarter second or so to insert a number? A lot of the efficiency of an insertion sort depends on the speed with which the actual insertion is made. In this little BASIC program, we used a FOR/NEXT loop (lines 45 through 55) to do the insertion. (Note that this would be way too slow if we were trying to do a couple of thousand insertions.) Luckily, in most languages, there are faster methods. But, in any case, the sorting method is not the important part of this month's discussion.

Now suppose, that instead of inserting a single number (as we were doing here), we were working with an entire set of information. Consider a typical mailing list, where we would be shuffling around a name, address, city, state, zip code, phone number, and various other bits and pieces. Can we, using BASIC, manipulate this information as easily as we sorted those numbers? Not quite, but we can come close.

Setting The Record

Take a look at this example of a Pascal record as I presented it last month:

TYPE

```
Cust_Rec = RECORD
  Name : String[30];
  Addr : String[30];
  City : String[15];
  State : String[2];
  Zip : 0..99999;
  Credit : ( OK, Avg, Bad );
END;
```

VAR

```
Mail_List : ARRAY [1..100]
  OF Cust_Rec;
```

Our variable **Mail_List** is an array of records, and each record holds several pieces of information about a given customer. Using the information in these records is almost easy. For example, we could find the zip code of customer number 17 by simply coding

```
Write( Mail_List[ 17 ].Zip )
```

The conversion from a number-sorting program to a record-sorting program is almost a trivial exercise in Pascal. While we can't duplicate the feat as easily in BASIC, we can at least simulate this convenient grouping of related pieces of information into a record. Again, look at Program 2. This is actually the same program as Program 1, but it uses strings to simulate records. If you look at the code from line 300 to line 410, you should be able to find a direct correlation to the statements of lines 30 to 60 of the first listing. True, the lack of string arrays in Atari BASIC has forced us to use some pretty strange looking assignment statements because we are now moving around substrings instead of simple numbers. I've tried to make these movements as clear as possible, but don't feel bad if it takes you some time to understand what is going on. I encourage you to print out the various strings (such as MAILLIST\$ and RECORD\$) at several points to see what is happening.

You may have noticed that these records are sorted based on the name of the person. Try this puzzle before reading on: Can you suggest ways of insuring that the sort is by zip code, instead?

And, if you have ST BASIC, Atari Microsoft BASIC, or OSS's BASIC XL or BASIC XE, you might try converting this program to use string arrays. I think you'll find that

the only real savings in coding complexity occurs in the actual insertion loop (lines 300 to 410). For the rest of the program, good old Atari BASIC doesn't suffer too much in comparison.

What have we accomplished? I hope you can see how, by isolating the record build/retrieve in subroutines such as those at lines 800 and 900 in this example, pseudo records are quite possible in BASIC. But we have also seen that the manipulation of these records can be tedious. And certainly moving all that string data around is not the fastest set of operations in the world. How could we improve things? Time to borrow some more concepts from structured languages such as Pascal: pointers and linked lists. But, for a look at those topics, we'll have to wait until next month.

Thought I forgot the answer to my little puzzle? Nope. Two ways to sort by zip code: Rearrange the order of the data in the RECORD\$ string so that the zip code comes first; or, change the master record comparison in line 340 so that only the zip code portions of the strings are compared. For example:

```
340 IF RECORD$(78,82) > MAILLIST
    $(RECPTR + 78,RECPTR + 82)
    THEN NEXT RECNUM:STOP
```

Program 1: Simple Numeric Insertion Sort

```
CO 10 DIM NPOS(20)
LK 15 FOR TOP=0 TO 19
EA 20 PRINT "GIVE ME A NUMBER
    BIGGER THAN 0 ";
IA 25 INPUT NPOS:IF NPOS<=0
    THEN 80
JD 30 IF TOP=0 THEN CHK=1:GO
    TO 60
CJ 35 FOR CHK=1 TO TOP
PF 40 IF NPOS>NPOS(CHK) THEN
    NEXT CHK:GOTO 60
JC 45 FOR MV=TOP+1 TO CHK STEP
    -1
BI 50 NPOS(MV)=NPOS(MV-1)
EM 55 NEXT MV
EK 60 NPOS(CHK)=NPOS
JN 65 NEXT TOP
ME 70 REM IF 20 NUMBERS, FALL
    THROUGH
FE 80 REM TO HERE WHEN 0 OR
    LESS ENTERED
DN 85 FOR CNT=1 TO TOP
BM 90 PRINT NPOS(CNT)
JC 95 NEXT CNT
```

Program 2: Insertion Sort of Pseudo-Records

```
AI 100 REM DATA DECLARATIONS
ME 110 DIM NAME$(30),ADDR$(3
    0),CITY$(15)
OL 120 DIM STATE$(20),ZIP$(5)
    ,CREDIT$(1)
```

```
EJ 130 RECSIZE=30+30+15+2+5+
    1:MAXREC=100
LI 140 DIM RECORD$(RECSIZE)
ED 150 DIM MAILLIST$(RECSIZE
    *MAXREC)
MH 160 DIM SPACES$(RECSIZE),Y
    ESNO$(1)
FK 170 SPACE$=" ":SPACE$(REC
    SIZE)=" "
LK 180 SPACE$(2,RECSIZE)=SPA
    CE$
FO 190 MAILLIST$=SPACE$:TOPR
    EC=0
CC 200 REM DATA ENTRY
BC 210 FOR TOPREC=TOPREC TO
    MAXREC-1
KE 220 GRAPHICS 0:PRINT TOPR
    EC;" CUSTOMERS IN FILE"
MJ 230 PRINT "ENTER ANOTHER
    CUSTOMER (Y/N) ";
OG 240 INPUT YESNO$:IF YESNO
    $="N" THEN 500
AF 250 GRAPHICS 0:GOSUB 700:
    REM ENTER A RECORD
NO 260 GRAPHICS 0:GOSUB 600:
    REM SHOW THAT SAME RE
    CORD
HE 270 PRINT:PRINT "IS THIS
    OKAY";
DE 280 INPUT YESNO$:IF YESNO
    $<>"Y" THEN 250
BO 290 GOSUB 900:REM CONVERT
    TO RECORD FORMAT
PH 300 REM FIND INSERT POINT
EI 310 IF TOPREC=0 THEN RPTR
    =0:GOTO 400
DA 320 FOR CHK=1 TO TOPREC
NP 330 RPTR=(CHK-1)*RECSIZE
AF 340 IF RECORD$>MAILLIST$(
    RPTR+1,RPTR+RECSIZE)
    THEN NEXT CHK:RPTR=RP
    TR+RECSIZE:GOTO 400
BA 350 REM INSERT RECORD
OP 360 FOR R=TOPREC TO CHK S
    TEP -1
FI 370 TEMP2=R*RECSIZE:TEMP1
    =TEMP2-RECSIZE
NL 380 MAILLIST$(TEMP2+1,TEM
    P2+RECSIZE)=MAILLIST$(
    TEMP1+1)
CN 390 NEXT R
OA 400 MAILLIST$(RPTR+1,RPTR
    +RECSIZE)=RECORD$
KB 410 NEXT TOPREC
HJ 500 REM
KA 510 REM DONE
ID 520 FOR RECNUM=0 TO TOPRE
    C-1
CG 530 RPTR=RECNUM*RECSIZE
DB 540 RECORD$=MAILLIST$(RP
    TR+1)
LN 550 GRAPHICS 0:GOSUB 800:
    GOSUB 600
BF 560 PRINT:PRINT "HIT RET
    URN TO SHOW NEXT RECO
    RD";
NO 570 INPUT YESNO$
KG 580 NEXT RECNUM
GJ 590 GOTO 200
IK 600 REM SUBROUTINE
LM 610 REM SHOW A RECORD
HO 620 PRINT "NAME ::";NAME$
HH 625 PRINT "ADDR ::";ADDR$
KP 630 PRINT "CITY ::";CITY$
EE 635 PRINT "STATE::";STATE
    $
CE 640 PRINT "ZIP ::";ZIP$
PJ 645 PRINT "CREDIT RATING
    (A TO F) ::";CREDIT$
HP 690 RETURN
IL 700 REM SUBROUTINE
AN 710 REM INPUT A RECORD
```

```
MJ 720 PRINT "NAME >";:INPUT
    #16,NAME$
MC 725 PRINT "ADDR >";:INPUT
    #16,ADDR$
PK 730 PRINT "CITY >";:INPUT
    #16,CITY$
IP 735 PRINT "STATE>";:INPUT
    #16,STATE$
SP 740 PRINT "ZIP >";:INPUT
    #16,ZIP$
OF 745 PRINT "CREDIT RATING
    (A TO F) >";
MB 750 INPUT #16,CREDIT$
IA 790 RETURN
IM 800 REM SUBROUTINE
BK 810 REM TAKE APART A RECO
    RD
AE 825 NAME$=RECORD$
KP 830 ADDR$=RECORD$(31)
NF 835 CITY$=RECORD$(61)
BP 840 STATE$=RECORD$(76)
JI 845 ZIP$=RECORD$(78)
FI 850 CREDIT$=RECORD$(83)
IB 890 RETURN
IN 900 REM SUBROUTINE
OO 910 REM BUILD A RECORD
EL 920 RECORD$=SPACE$
DB 925 RECORD$(1,30)=NAME$
EC 930 RECORD$(31,60)=ADDR$
GO 935 RECORD$(61,75)=CITY$
LK 940 RECORD$(76,77)=STATE$
CP 945 RECORD$(78,82)=ZIP$
PA 950 RECORD$(83,83)=CREDIT
    $
IC 990 RETURN
```

©

≡CAPUTE!≡

INSIGHT: Atari

The code that appears in the October 1987 "INSIGHT: Atari" column is correct as listed, with one minor change. Just before the last line (.END or END), the variable SNAME needs to be declared. The proper declaration is
SNAME.BYTE "S;"

Amiga Marbles

This program, from the October 1987 issue, is correct as listed, but it needs the graphics.bmap file on your Extras disk. If you are missing this file, the 1.2 Extras disk contains a program called ConvertFD which will create it for you. Run the program and enter Extras:fd-1.2/graphics_lib.fd for the file to convert. Enter graphics.bmap for the output file. When you run Marbles, this file must either be copied to the current directory or the LIBS directory on your boot disk; otherwise, Amiga Basic will stop with a file not found error. ©



That Month Again

Amazing Product Rallies Information Lunatics

By now, most of you have heard that Atari has announced that it is, indeed, going to sell a CD-ROM. The advantage of a CD-ROM is that a single optical disk can hold hundreds of megabytes of information. The disadvantage is that CD-ROMs are exactly what the second part of their acronyms suggest: Read Only Memory. The computer can not write to such a device.

But the computer industry is working very hard to overcome this restriction. Welcome to the world of the WORN—Write Once Read Many. Special optical disk drives have already been introduced that use lasers to write information. The data thus written cannot be changed, but it can effectively be “erased” and a later, updated copy can be written to another part of the disk. A typical home user could probably use a single such optical disk for a couple of years before needing to copy the most recent versions of all files to a new, clean disk. But don’t hold your breath waiting to buy one—at least not unless you’d rather buy one than, say, a new sports car. However . . .

Fantastic Option Overlooked!

I know it may be hard to believe, but the designers of the original eight-bit Atari computers, way back in 1979, included a close relative of the WORN in their design. True, it is slower than a WORN, and it isn’t as easy to use, but it works! And yes, the WORN is built into each Atari eight-bit computer!

There are a couple of ways to use an Atari WORN, but here is one of the simplest. From BASIC, just type in the command:

POKE 803,87

Then load a BASIC program

type:

SAVE “WORN:TEST”

Presto! Your program will be saved to this marvelous device. (Hit RESET to disable the WORN.)

Of course, you should be careful not to rely on the WORN. Certainly, compared to a WORM, recovering programs saved to this Write Once Read Never device can take a while. If you happen to have a LAND device handy, you can make a quick copy of small programs saved to the WORN, but otherwise you will probably have to ensure a reliable connection between your biological optical devices and your digital extremity input devices.

WYSIWYG

Another marvelous acronym, pronounced “wizz-ee-wigg,” is an old one that is relatively new to computers: What You See Is What You Get. Usually applied to word processing programs, where it means that the printed copy will look like the screen display (implying a higher-resolution display than that of an eight-bit Atari), this time I use it in its old meaning, the one a flea market vendor might use. Take another look at just the initial letters of the words in my headings up until now. Together they make a single acronym. One very appropriate to this month’s issue.

Actually, my tale of the WORN device owes much to tales of WOM (Write Only Memory) devices that have abounded in computer folklore for ages. (Well, 10 or 15 years is “ages” when it comes to computers, right?) I remember one article that showed a picture of a water tower and claimed it was a WOM big enough for a whole town. So, if you don’t like jokes, I apologize, but I haven’t pulled an April Fool jest in a couple of years. It was time. (Oh, yes, the LAND above is not an

acronym: I was referring to a Polaroid Land camera. And *biological optical devices* are your eyes, and *digital extremity input devices* are your fingers, of course.)

Without Honor?

A couple of my columns lately have turned out to be mildly prophetic of other COMPUTE! articles. One article that related to some of my recent comments was “Tri-Sort for Atari” on page 88 of the February issue, in which Arthur Horan provides you with a fast machine language sort that you can use with the pseudofields and pseudorecords I described in my February and March columns. The Shell-Metzner sort used by Mr. Horan is not the fastest for very large arrays of data, but it is probably quite well suited for the number of records you can pack into an Atari BASIC string. In my March column (which, of course, was written long before I saw the February issue) I said that I hoped you wouldn’t use my quick-and-dirty bubble sort. With the help of Mr. Horan, you don’t have to.

Last month, I also promised to return to the subject of my December article: Acrostic and other word puzzles. Well, in the December issue I said that I had yet to see a really good crossword puzzle program. Lo and behold, on page 61 of the February issue is a review of *Crossword Power* (for IBM PCs) that shows indeed how limited such programs are. I think the program did a creditable job with the number of words it was given, but the result was far from ideal.

For example, a typical newspaper crossword puzzle is perhaps 5–10-percent black space. The one shown in that review was more like 75-percent black space. Too, it is considered less than ideal for words in a newspaper puzzle to have more than one uncrossed letter. In the puzzle of the review, several

words are "hooked in" by a single letter! In at least one case, this results in a clue with two answers. (See 21 Across: A musical instrument. Is it a piano or cello?) Granted, the reviewer gave the program very few words to work with (only 35), but I can't help but wonder how long it would take to generate a good puzzle if one gave it a list of a couple of thousand words.

More Words About Words

In this same vein, several readers wrote to give comments and suggestions about the acrostics problem. (To refresh your memory: The problem is to write a program that will produce all valid five-by-five acrostics or word squares from a given list of five-letter words. Assume that there are 5000 words in the list.) One gentleman suggested that I was making the problem too hard: I should limit the number of words and accept the first puzzle produced. Well, yes, that wouldn't take as long, but that is kind of like building a chess-playing program that can only take over after a human has played the first 40 moves, and even then it can only play until it finds the first check (but not mate). As a practical matter, perhaps the gentleman is right. As a mathematician (which I was, once, I think), I want to see a problem solved, not sidestepped.

I even got two versions for other computers. An Amiga version took about three times as long on the Amiga as on the eight-bit Atari. But that is because of the inefficient way that Microsoft BASIC strings are implemented.

As for myself, I haven't had time to put together a complete solution, but I have started a couple of paper designs. I am convinced that, as with so very many computer problems, a really good solution depends on finding the right way to represent the data (in this case, the word list).

One possibility is this: How about a "map" wherein every single possible five-letter word is represented by a YES/NO flag? (That is, yes or no that the flagged word exists in our word list.) In compact form, such a map requires 26^5 bits, or about 1.5 megabytes. In a more practical form (use a 32-bit com-

puter word for each set of 26 bits), one still needs just a little under 1.9 megabytes. Hmmmm . . . anybody with a four-megabyte ST listening out there? (Actually, for efficiency, you would want four maps of increasing size— 26^2 , 26^3 , 26^4 , and 26^5 —to represent the possible sequential letter sets. With some intelligent compression, all this might be possible in a half megabyte or so.)

I also tend to think that building the valid word set via a linked tree or list would work (albeit probably slower than the brute force approach, above). At worst, such a list would need about 75,000 bytes. Given the likely letter patterns in 5000 English words, I wouldn't be surprised to find that we could make do with 30,000 bytes or fewer. (Now we're down in eight-bit territory again!)

Are you asking "What is a linked list?" That's a big topic. For now, let me show you a way of simulating a word tree in Atari BASIC. The accompanying listing looks long, but you will quickly find that the bulk of it is nothing but simple DATA statements. This program has no real practical value, so don't feel that you need to type it in unless you are curious. But I do hope that at least some of you will look at my word tree and become inspired. If you are, write to me (P.O. Box 710352, San Jose, CA, 95171-0352).

Word Tree

For instructions on entering this program, please refer to "COMPUTE!'s Guide to Typing In Programs" elsewhere in this issue.

```

DE 100 DIM COUNT(3), LINE(3),
    MAX(3)
PC 110 DIM WORD$(3), LETTER$(
    1)
DE 120 GRAPHICS 0
KA 200 LEVEL=0: LINE(LEVEL)=1
    000
NK 220 STOP
DP 300 REM RECURSIVE SUBROUT
    INE
KJ 310 RESTORE LINE(LEVEL)
JH 320 READ MAX
DP 330 LEVEL=LEVEL+1
DD 340 COUNT(LEVEL)=1: MAX(LE
    VEL)=MAX
II 350 RESTORE LINE(LEVEL-1)
    +COUNT(LEVEL)
PN 360 READ LETTER$, LINE
CI 370 WORD$(LEVEL, LEVEL)=LE
    TTER$: LINE(LEVEL)=LIN
    E
NL 380 IF LINE=0 THEN PRINT
    WORD$
DP 390 IF LINE<>0 THEN GOSUB
    300

```

```

HF 400 COUNT(LEVEL)=COUNT(LE
    VEL)
BF 410 IF COUNT(LEVEL)<=MAX(
    LEVEL) THEN 350
CD 420 LEVEL=LEVEL-1
NH 430 RETURN
NL 1000 DATA 8, (FIRST LETTE
    R)
AL 1001 DATA A, 1100
AO 1002 DATA B, 1200
BO 1003 DATA C, 1300
DH 1004 DATA L, 1400
CA 1005 DATA N, 1500
CB 1006 DATA O, 1600
CB 1007 DATA P, 1700
CH 1008 DATA T, 1800
AM 1100 DATA 1, (SECOND LETTE
    R, A*)
BO 1101 DATA R, 1110
BC 1110 DATA 1, (THIRD LETTER
    S, A*)
HF 1111 DATA E, 0
AC 1200 DATA 1, (SECOND LETTE
    R, B*)
DD 1201 DATA E, 1210
AH 1210 DATA 1, (THIRD LETTER
    S, B*)
IP 1211 DATA T, 0
AF 1300 DATA 2, (SECOND LETTE
    R, C*)
DD 1301 DATA A, 1310
CB 1302 DATA O, 1320
AH 1310 DATA 3, (THIRD LETTER
    S, C*)
IK 1311 DATA N, 0
IN 1312 DATA P, 0
JC 1313 DATA T, 0
DE 1320 DATA 1, (THIRD LETTER
    S, C*)
JJ 1321 DATA T, 0
AP 1400 DATA 2, (SECOND LETTE
    R, L*)
DD 1401 DATA A, 1410
CD 1402 DATA O, 1420
AP 1410 DATA 1, (THIRD LETTER
    S, L*)
IN 1411 DATA P, 0
BO 1420 DATA 1, (THIRD LETTER
    S, L*)
IO 1421 DATA P, 0
DD 1500 DATA 1, (SECOND LETTE
    R, N*)
DJ 1501 DATA E, 1510
DB 1510 DATA 1, (THIRD LETTER
    S, N*)
JC 1511 DATA T, 0
DD 1600 DATA 1, (SECOND LETTE
    R, O*)
CI 1601 DATA R, 1610
CF 1610 DATA 1, (THIRD LETTER
    S, O*)
IE 1611 DATA E, 0
DB 1700 DATA 2, (SECOND LETTE
    R, P*)
DH 1701 DATA E, 1710
CJ 1702 DATA O, 1720
BL 1710 DATA 2, (THIRD LETTER
    S, P*)
IO 1711 DATA N, 0
BL 1800 DATA 2, (SECOND LETTE
    R, T*)
DL 1801 DATA A, 1810
CL 1802 DATA O, 1820
DH 1810 DATA 3, (THIRD LETTER
    S, T*)
ID 1811 DATA B, 0
JA 1812 DATA N, 0
JD 1813 DATA P, 0
CL 1820 DATA 2, (THIRD LETTER
    S, T*)
JA 1821 DATA N, 0
JD 1822 DATA P, 0

```




More On Structure

In last month's column, I discussed structured data types. In the program listing, I then used a large string and its substrings to simulate an array of records. This month, I will continue my efforts to help you write more structured programs, but first I'll make some general comments on Atari BASIC strings.

I think that last month's program demonstrates that long strings can be just as powerful as string arrays. I would expect that program to operate at least as quickly as an equivalent Microsoft BASIC program using string arrays, because the typical Microsoft implementation goes through a lot of overhead generating and reclaiming dynamic strings. In that example, we were inserting new records into our string structure. If we had been deleting records, Atari BASIC really would have shone. For example, suppose we have a string filled with 50-character pseudorecords. To delete the third such record, we could simply do this:

```
RECORD$(101) = RECORD$(151)
```

Presto! All records are moved up one spot, and the third one is gone.

A small sidetrack: Unfortunately, one failing of Atari BASIC is that it has no built-in way to conveniently save and restore such long strings to and from disk. The most common output method is to PRINT# such a string, but that doesn't help a lot, since INPUT# is limited to no more than 255 bytes per line. I have seen many users resort to using a FOR/NEXT loop to PRINT# or INPUT# the individual pseudorecords one at a time. That works, but it certainly slows down disk I/O speed. In BASIC XL and BASIC XE, we added a pair of special statements for this purpose (BPUT# and BGET#, where the B stands for *Block* or *Buffer*), but you can accomplish the same thing in Atari BASIC with a pair of fairly

short USR assembly language subroutines. (These routines have been published several times, and I won't repeat them this month.)

Out Of Sorts

To continue my discussion of how to achieve structured data features in Atari BASIC, I call your attention to Program 1. Study it, type it in, and run it. It is a fairly clumsy but working record-sort routine. That is, it sorts the kinds of pseudorecord strings that we also used in last month's program. (Incidentally, I have used the worst of all possible sort algorithms: the bubble sort. Please, if you are serious about sorting your data, learn a couple of other methods, such as the heap sort, Shell sort, and quicksort. Why do I use the bubble sort? Because it is the smallest and easiest to demonstrate. Or maybe because I'm just lazy.)

As you study that listing, pay attention to lines 230-260, where the tests and swaps necessary to any sort routine are made. Because I purposefully organized the data in my array of pseudorecords in the worst possible way (for a bubble sort, at least), the IF test will never branch around the swap, and we will make more than 4900 of these string swaps. Surely there must be a better way.

Pointing The Finger

Time to introduce another concept from the structured languages. In any computer language, moving blocks of data around (whether records, pictures, disk blocks, or whatever) is time-consuming. So most programs don't actually move the data. Instead, they move pointers to the data.

What is a pointer? Quite simply, a pointer is a variable that contains the address of another variable. Atari BASIC allows only one *explicit* kind of pointer—the ADR function that gives the address of a string. And, indeed,

many programmers use ADR as a pointer when they pass the address of a string to an assembly language subroutine. (Imagine having to pass the bytes of a string through a series of POKES.)

But there is another, hidden kind of pointer in almost any computer language: array and string indices. As an example, if the record data we are working on is *always* within a particular string, for example, then we need only know the relative position of a given record within the string in order to obtain its information. We most likely do not care about the actual physical memory address of the data.

Merge the lines shown in Program 2 to those already in place in Program 1 (deleting the three lines noted) and study the results of using implicit pointers. There is (rather obviously) little difference between the two programs. Instead of swapping actual substring pseudorecords, we are now swapping only the indices into the master string. When you run this second version, you should notice a speed improvement of almost 2:1. (I got 70.8 seconds versus 135.3 seconds, but that was done using the FAST mode of BASIC XL. Your times will likely be slower.)

As clever as this trick is, it does not answer all of a programmer's needs. Suppose, for example, that the data to be sorted resides in two or three different arrays. The logistics in BASIC get complicated. In a language with more data structuring capabilities, where a pointer can point to a given record type no matter where the record might be, such a division of data would probably make virtually no difference on program speed.

Enough for this month. Next month, we will go back to the acrostics puzzle of the December issue, since it generated more mail

for me than any topic in recent months. If I have managed to convince you that records and pointers are valuable tools, wait until you see my proposed solution to the acrostics problem!

Program 1

```

BD 100 DIM TEST$(10000),TEMP
      $(100),BLANK$(100)
NK 110 BLANK$=" ":BLANK$(100)
      )=" ":BLANK$(2)=BLANK
      $
NI 120 FOR R1=0 TO 99:RPTR=R
      1*100+1
GO 130 TEST$(RPTR)="THIS IS
      RECORD NUMBER "
HP 140 TEMP$=STR$(199-R1):TE
      MP$(4)=BLANK$
DM 150 TEST$(RPTR+22)=TEMP$(
      2)
FJ 160 NEXT R1
IF 170 REM OUT OF ORDER...SO
      RT THEM
LF 180 POKE 20,0:POKE 19,0
BP 190 FOR R1=98 TO 0 STEP -
      1
JA 200 FOR R2=0 TO R1
JC 210 PTR1=R2*100+1
NM 220 PTR2=PTR1+100
MC 230 IF TEST$(PTR1,PTR1+99
      )<TEST$(PTR2,PTR2+99)
      THEN 270
PJ 240 TEMP$=TEST$(PTR1,PTR1
      +99)
GO 250 TEST$(PTR1,PTR1+99)=T
      EST$(PTR2,PTR2+99)
PN 260 TEST$(PTR2,PTR2+99)=T
      EMP$
FM 270 NEXT R2
NA 280 PRINT "*";
FN 290 NEXT R1
CA 300 PRINT
BF 310 TICK=PEEK(20):TOCK=PE
      EK(19):IF TICK<>PEEK(
      20) THEN 310
EP 320 TIME=TICK+256*TOCK
GC 330 PRINT "THAT TOOK ";TI
      ME/60;" SECONDS"
CE 340 PRINT "HIT RETURN TO
      SEE LIST ";
IC 350 INPUT TEMP$
ND 360 FOR R1=0 TO 99:RPTR=R
      1*100+1
DF 370 PRINT TEST$(RPTR,RPTR
      +99)
FN 380 NEXT R1

```

Program 2

```

CJ 105 DIM RPT(99)
OH 125 RPT(R1)=RPTR
MC 210 PTR1=RPT(R2)
CA 220 PTR2=RPT(R2+1)
CA 230 IF TEST$(PTR1,PTR1+99
      )>TEST$(PTR2,PTR2+99)
      THEN RPT(R2)=PTR2:RP
      T(R2+1)=PTR1
CN 240 REM DELETE
PE 250 REM THESE
EL 260 REM 3 LINES!
AD 360 FOR R1=0 TO 99:RPTR=R
      PT(R1)

```

Cursor Plus

Emmanuel Gendrano and Greg Knauss

Add even more power to the Atari 400, 800, XL, and XE editor device. Compatible with most environments and programs.

Atari computers have the most powerful screen editing features of any eight-bit computer. "Cursor Plus" extends the editor even further, adding functions to move the cursor by one word, change the case of an entire screen line, delete the remainder of the line, and more.

Cursor Plus is compatible with most programs that use the E: editor device. This includes Atari BASIC, BASIC XL, MAC/65, and other environments. It *does not* include programs that use their own editors, such as *Action!*, *Atariwriter*, and Atari's *MEDIT* editor.

Typing It In

Cursor Plus works on all Atari eight-bit computers. Type it in and save it to tape or disk. The program is written in BASIC, but it creates a machine language program when it is run.

After saving the program, type RUN. You'll be asked if you want to save a copy or install it in memory. If you have a disk system, you must choose the option to save a copy. The program asks for a filename with which to save Cursor Plus. Be sure to use a name that's different from the one you used to save the creator program. After the machine language program has been written to disk, you can install Cursor Plus by going to DOS and using the L option to load it. Alternatively, you can name the file AUTORUN.SYS, causing the program to install automatically whenever you boot up the computer.

If you have a tape system, choose the memory option.

The New Editor

When you're using Cursor Plus, all of the regular cursor movement and editing capabilities are still in effect. In addition, Cursor Plus adds the following controls:

- **Control-ESC** Go to the end of the line. This command moves the cursor to the end of the physical screen line (not the logical line).
- **Control-Shift-RETURN** Go to the beginning of the line. This command moves the cursor to the beginning of the physical screen line. It does not enter the line, as a regular RETURN would.
- **Control-Shift-(minus sign)** Go to previous word. This command moves the cursor to the word immediately to the left of the current cursor position.
- **Control-Shift-(equals sign)** Go to next word. This command moves the cursor to the word immediately to the right of the current cursor.
- **Control-Shift-SPACE** Delete to end of physical line. This command deletes everything after the current cursor position to the end of the line.
- **Control-Shift-Caps** Change case of the rest of the physical line. This command begins at the current cursor position and changes all lowercase letters to uppercase and all uppercase letters to lowercase. Single words can be changed by cursoring to the beginning of the word, changing the case of the rest of the line, cursoring to the next word, and changing the case again.
- **Control-Shift-Inverse** Change regular text to inverted text and inverted text to regular text. This command is similar in operation to the change case command. It begins at the current cursor position and changes the high bit of all of the text to the end of the physical line, thus, changing inverted letters (blue on white) to normal letters



Machine Language Graphics: The Final Installment

This month I will finally show you the machine language equivalents of the most important BASIC screen I/O operations. We'll begin with an example. Suppose we wanted to implement a GRAPHICS 7 statement. From two months ago, we know that the equivalent low-level statements are

```
CLOSE #6
OPEN #6,12+16,7,"S:"
```

A direct translation into machine language follows.

```
;GRAPHICS 7
;CLOSE IOCB 6
LDX #$60          ; IOCB number
LDA #12           ; the CLOSE command
STA ICCOM,X       ; put in place
JSR $E456         ; call CIO
;OPEN IOCB 6
LDX #$60          ; IOCB number
LDA #3            ; the OPEN command
STA ICCOM,X       ; put in place
LDA #12+16        ; give it the same
                  ; value
STA ICAX1,X       ; as you would in
                  ; BASIC
LDA #7
STA ICAX2,X
LDA #DEVICE&$FF   ; don't worry why
STA ICBAL,X       ; this works
LDA #DEVICE/$100 ; it just does.
STA ICBAL+1,X
JSR $E456         ; do the real work
```

Don't bother assembling this code yet—it won't work without some of the help given later in this article.

Now, if you all you ever wanted to do was emulate GRAPHICS 7, that would be an adequate method. But in BASIC, the general form of the command is GRAPHICS *mode*, where mode is any numeric variable or expression or your choice. It would be better if we could emulate *that* in machine language. And, to some degree, we can.

In BASIC's GRAPHICS statement, the mode value is called a *parameter* to the operation. In machine language, we also use parameters. With the 6502 microprocessor that Atari machines use, we usually try to pass the parameters in one or more of the three registers that the chip possesses: the A register (also called the accumulator), the X register, and the Y register. Suppose you need to pass an IOCB number. Since it needs to be in the X register for the call to CIO anyway, why not pass it there?

The listing that follows is *not* a program in and of itself. Rather it is a set of subroutines that your program may call (via JSR) to implement the given operation. At the very end of the article you will find a sample program that calls these subroutines.

When you use the subroutines in your own programs, you must note carefully the description of the parameters that I have given. Be sure that the appropriate registers contain the proper values before you jump to a subroutine.

The listing is given without line numbers. Some assemblers use line numbers, but, when they do, it's for editing purposes only—the numbers have no effect on the program. Comments are preceded by a semicolon. You may omit any of them that you like. When you have typed all this in (and have checked it *carefully* for errors—one mistake can cause a lockup), you should save it (or LIST it, depending upon your assembler) to disk or tape. You can then use it as the nucleus of your own graphics programs.

```
;
; Equates
;
; Without these, the program won't assemble properly
;
ICCOM = $342 ; the COMMAND byte in the IOCB
ICBAL = $344 ; the low byte of the buffer address (filename)
ICBL = $348 ; the low byte of the buffer length
ICAX1 = $34A ; auxiliary byte 1: type
ICAX2 = $34B ; auxiliary byte 2: mode
;
CIO = $E456 ; Central Input/Output routine
ROWCRS = 84 ; ROW CuRSor—y position
COLCRS = 85 ; COLUMN CuRSor—x position
ATACHR = 763 ; where line color goes for DRAWTO
;
; Now the working routines
;
; REMEMBER: these are only subroutines
; You must call them via JSR from your own code
;
; CLOSE channel
;
; Parameter: X register holds IOCB number
; On exit: Y register holds error code
;
CLOSE
LDA #12          ; close command
STA ICCOM,X      ; in place
JMP CIO          ; do the real work
;
; OPEN channel,type,mode,file
;
; Parameters: X register holds IOCB number
; A register holds type
; Y register holds mode
; the address of the file/device
; name must already be set up
; in the IOCB—
; On exit: Y register holds error code
;
OPEN
STA ICAX1,X      ; the type value
TYA
```



```

STA ICAX2,X      ; and the mode, if appropriate
LDA #3           ; OPEN command
STA ICCOM,X      ; in place
JMP CIO          ; the real work

```

```

;
; GRAPHICS mode

```

```

; Parameter: A register holds desired mode
; On exit:   Y register holds error code

```

```

; GRAPHICS

```

```

PHA              ; save the mode for a moment
LDX #$60         ; always use IOCB #6
JSR CLOSE        ; be sure it is closed
LDX #$60         ; the same IOCB again
LDA #SNAME&$$FF ; the "S:" device name
STA ICBAL,X      ; must be put in place
LDA #SNAME/$100 ; before we go further
STA ICBAL+1,X    ; (take this part on faith)
PLA              ; recover the GRAPHICS mode
TAY              ; put it where OPEN wants it
AND #16+32       ; isolate the text window and no-clear bits
EOR #16          ; flip state of the text window bit
ORA #12          ; allow both input and output
JMP OPEN         ; do this part of the work

```

```

;
; PUT channel,byte

```

```

; Parameters: A register holds byte to output
;             X register holds channel number
; On exit:   Y register holds error code

```

```

PUT
TAY              ; save the byte here for a moment
LDA #0
STA ICBLL,X      ; $0000 to length
STA ICBLL+1,X    ; as noted last month
LDA #11          ; the command value
STA ICCOM,X
TYA              ; data byte back where CIO wants it
JMP CIO

```

```

;
; byte = GET( channel )

```

```

; Parameter: X register holds IOCB number
; On exit:   A register holds byte from GET call

```

```

GET
LDA #0
STA ICBLL,X      ; $0000 to length...
STA ICBLL+1,X    ; as noted last month
LDA #7           ; the command value
STA ICCOM,X      ; where CIO wants it
JMP CIO          ; believe it or else, that's all

```

```

;
; PLOT x,y,color

```

```

; Parameters: A register holds color
;             X register holds x location
;             Y register holds y location
; NOTE: not for use with GR.8 or GR.24

```

```

PLOT
STX COLCRS       ; see my August column
STY ROWCRS       ; these are just POKes
LDX #$60         ; the S: graphics channel
JMP PUT          ; color is already in A

```

```

;
; byte = LOCATE( x,y )

```

```

; Parameters: X register holds x location
;             Y register holds y location
; On exit:   A register holds color of point at (x,y)

```

```

LOCATE

```

```

STX COLCRS       ; again, see column
STY ROWCRS       ; from two months ago
LDX #$60         ; the S: graphics channel
JMP GET          ; color returned in A

```

```

;
; DRAWTO x,y,color

```

```

; Parameters: A register holds color
;             X register holds x location
;             Y register holds y location
; NOTE: not for use with GR.8 or GR.24

```

```

; DRAWTO

```

```

STX COLCRS       ; once more: see the article
STY ROWCRS       ; from two months ago
STA ATACHR       ; location 763, also in that article
LDX #$60         ; again, we use IOCB #6
LDA #17          ; the XIO number for DRAWTO
STA ICCOM,X      ; is actually the command number
JMP CIO          ; and that's all we really need to do

```

Now that we have these routines, how do we use them? A full explanation would need at least the beginnings of a tutorial book. But here's a short example. First, a small program in BASIC:

```

100 GRAPHICS 3+16
110 COLOR 2 : PLOT 10,10
120 COLOR 3 : PLOT 20,20
130 COLOR 1 : PLOT 0,15 : DRAWTO 30,15
140 GOTO 140 : REM (just wait for RESET)

```

Now the same thing in machine language, using the routines of the program above. The only decision you will have to make is where in memory to place the assembled code. My first line reflects what should be a safe choice for most assemblers used in most 48K byte (or 64K byte) machines. If your assembler has a SIZE or MEMORY command, use it to get an idea of what is safe. In any case, LIST or SAVE your code to disk or tape before assembling, just in case.

```

                *= $6000      ; my "usually safe" location
;
START LDA #3+16      ; first
JSR GRAPHICS        ; emulate GRAPHICS 19
;
LDX #10             ; now do PLOT 10,10
LDY #10             ; (x and y locations)
LDA #2              ; with COLOR 2, a slight
JSR PLOT            ; change from BASIC, but close
;
LDX #20             ; similarly:
LDY #20             ; we want PLOT 20,20
LDA #3              ; with COLOR 3
JSR PLOT            ; one call does it all
;
LDX #0              ; last PLOT:
LDY #15             ; PLOT 0,15
LDA #1              ; with COLOR 1
JSR PLOT            ;
;
LDX #30             ; and now the DRAWTO:
LDY #15             ; DRAWTO 30,15
LDA #1              ; still with COLOR 1
JSR DRAWTO          ; the routine does the work
;
LOOP1 JMP LOOP1      ; (loop here until RESET is pressed)
;
; Append all of the code for all of ; the subroutines here
;
; Your assembler may need .END or END as
; the very last line

```

